CS337: Artificial Intelligence and Machine Learning

Krishna Agaram

Autumn 2023

Contents

Introduction

These are course notes made for a beginner course in AI/ML (CS337) offered in Fall'23 at IIT Bombay.

The course is primarily focused on Machine Learning. <Topics covered>

Course logistics

Class hours: (LH102) Monday 9.30AM Tuesday 10.30AM Thursday 11.30AM

Main platform of communication: Moodle.

Grading

- Participation: 5%
- Best of two in-class Quizzes: 15%
- Midsem examination: 25%
- Final examination: 40%
- Course project: 15%

Course project

The course project is a group project. Each group will have 3-4 students.

Use techniques studied in this course to solve an interesting problem. The paper chosen should be reproduced, at least. Further improvements are encouraged. A nontrivial paper must be chosen.

Lab Logistics

Compulsory attendance. Grading:

- Weekly labs: 35%
- Quiz during the midsem week: 15%
- Final exam 50%

The labs are **autograded**.

Chapter 1

Lecture 1

1.1 Machine Learning

1.1.1 What is the buzz about?

In a nutshell, Machine Learning is:

- Well, the ability of machines to learn from data or past experience.

What's *data/past experience*? These are *numbers* that come from various sources such as sensors, domain knowledge, experimental runs, etc.

What's *learn*: Make accurate predictions or decisions based on data by optimizing a model. Here we just cannot miss:

All Models are Wrong, but Some Are Useful
- George E. P. Box

Machine Learning and Mathematical Statistics are closely related, and there is a lot of overlap.

1.1.2 A brief history of Machine Learning

- 1950s: Neural Networks and Perceptron
- 1960s: Decision Trees
- 1970s: Nearest Neighbors
- 1980s: Support Vector Machines, and after this the AI winter started and lasted for 10 years.
- 1990s: Boosting, Convolutional Neural Nets
- 2000s: Deep Learning

1.1.3 When do we need ML?

ML is **not** the universal sledgehammer. One typically uses ML for tasks that are easily performed by humans but are complex for computer systems to emulate. For example, recognizing a face, understanding speech, etc. Most tasks that humans solve using human intelligence - hence the name Artificial Intelligence: can a machine solve problems that intelligent humans can?

Computer Systems are really good at solving problems that can be encoded as a set of rules. Many tasks that humans perform effortlessly are difficult to encode as a set of rules - we can do it, but we do not perfectly know how we do it; but one thing is for sure: we learnt it from experience. These are the kind of situations where ML is useful.

A (non-exhaustive) set of problem styles where ML is useful:

- **Vision**. Identity faces in a photograph, objects in a video or image etc.
- **Natural Language**. Translate a sentence from one language to another, answer questions - chatbots, understand the sentiment of a sentence, etc.
- **Speech**. Recognize spoken words, speak sentences naturally.
- **Game playing**. Play games like chess, go, etc as good or better than expert humans.
- **Robotics**. Make robots (computers) that can navigate in the real world, pick up objects, etc.
- And **more**. Self-driving cars, maze navigation, etc.

There is another domain of problems where we require ML: Solving problems that are beyond human intelligence; for example, analyzing a ton of data(knowledge) - large complex *datasets*. For example, IBM's `Watson` https://www.sciencedirect.com/science/article/pii/S0004370212000872 beat the world's best players at Jeopardy. It did this by analyzing a large amount of data - Wikipedia, etc.

There has been remarkable success in ML over the years. This is because of the availability of large datasets - because of the internet, and the availability of fast computers - because of Moore's law.

1.2 Types of Machine Learning

1.2.1 Supervised Learning

Here is the general problem:

Definition 1.1 (Dataset). A dataset is a collection $\mathcal{D}$ of examples. Each example is a pair of *input* $x \in \mathcal{X}$ and *output* $y \in \mathcal{Y}$. We write

$$\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\} = \{(x_i, y_i)\}_{i=1}^{n}$$

The set $\mathcal{X}$ is called the feature space and $\mathcal{Y}$ the label space.

When $\mathcal{Y} = \{0, 1\}$, we call the problem a binary classification problem. When $\mathcal{Y} = \{1, \ldots, k\}$, we call the problem a multi-class classification problem. When $\mathcal{Y} = \mathbb{R}$, we call the problem a regression problem.

Definition 1.2 (Supervised Learning). Let the instances in $\mathcal{D}$ be drawn from some unknown (true) distribution P over $\mathcal{X} \times \mathcal{Y}$. We want to find a function $h : \mathcal{X} \to \mathcal{Y}$ such that for a new test instance (x, y), $h(x) = y$ with high probability., - $f(x)$ is a good predictor of y for any $(x, y) \sim P$. We call h a hypothesis or a model.

Examples of supervised learning problems:

- Given a set of images of cats and dogs, predict whether a new image is a cat or a dog.

- Given a set of images of handwritten digits, predict the digit in a new image.
- Given some data about the human body, predict the height of a person.

Techniques in supervised learning: Decision Trees, Neural Networks, Support Vector Machines, etc.

1.2.2 Unsupervised Learning

Unsupervised learning is the problem of finding patterns in data - we do not have any predefined labels. The problem is to find some structure in the data. For example, given a set of images of cats and dogs, we might want to find out if there are any subgroups of cats and dogs.

Techniques in unsupervised learning: Clustering, PCA, mixture models, etc.

1.2.3 Reinforcement Learning

This is a different paradigm that encourages models to learn by actively interacting with the environment. There is no specific dataset, but the model learns by trial and error. **Rewards** (negative rewards are called punishments) are given to the agent for taking actions that are not good in the short run.

Definition 1.3. The goal in Reinforcement Learning is for the agent to maximize **expected** total (or cumulative) reward.

The programmer decides these rewards as required for the application, to guide the agent towards mastery in the application. For example, a robot learning to play chess.

Techniques in reinforcement learning: Q-learning, policy gradient, etc.

1.3 The ML pipeline

1. **Data Collection**. Collect data for your problem. The quality of the data determines the quality of the model.
2. **Representation**. Choose features that represent your data. Raw? Expert-derived? Learned?
3. **Modelling**. Choose a model that can learn from the data. Linear? Nonlinear? What are the computationsl overheads? The model you choose is called the hypothesis class $\mathcal{H}$.
4. **Training**. Learn the parameters of the model from the data. This is called fitting the model to the data. Essentially, one finds the hypothesis $h \in \mathcal{H}$ that works best. To define 'works best', one needs to define an objective and work towards maximizing/minimizing the objective.
5. **Prediction/Inference**. Evaluate the model on a testset - that is, assign labels to unseen data. Choose an evaluation metric to quantify performance on testset.

Chapter 2

Lecture 2

2.1 Linear Regression

Let us consider the supervised learning problem where

$$\mathcal{X} = \mathbb{R}^d$$
$$\mathcal{Y} = \mathbb{R}$$

that is, the output is a real quantity and not a classification. The linear model has been a mainstay of statistics, perhaps because of its simplicity to analyse.

We predict the output using a linear model: That is, each predictor h is a **linear** function of the input $x \in \mathbb{R}^d$:

$$h(x) = w_0 + w_1 x_1 + \cdots + w_d x_d = \mathbf{w}^\mathbf{T}\mathbf{x}$$

where

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_d \end{pmatrix}, \mathbf{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_d \end{pmatrix}$$

We write $h = h_w$. The class $\mathcal{H}$ is then $\{h_w : w \in \mathbb{R}^{d+1}\}$. Typically, 1 is included in the input vector $\mathbf{x}$ (and the dimension with 1 is called d).

In $(d+1)$-dimensional input-output space, the linear model $\hat{y} = \mathbf{w}^\mathbf{T}\mathbf{x}$ is a d-dimensional hyperplane. Why? Since the element $\tilde{x} = (\mathbf{x}, \hat{y}) = (1, x_1, \ldots, x_{d-1}, y)$ is in the nullspace of matrix $A = \begin{bmatrix}\mathbf{w} & -1\end{bmatrix}^\mathsf{T}$ and so is a subspace, hence hyperplane, of dimension d ($\text{rank}(A) = 1$).

We would like each datapoint in $\mathcal{D}$ to lie best-case on, and preferably *close to* this hyperplane - definitions for closeness, as we shall see next, are motivated by this geometric interpretation.

There are two questions left to answer:

1. *How do we measure closeness? That is, what is the loss function?* A loss/error function $\mathcal{L}(h, \mathcal{D})$ is a measure of performance of the predictor h on dataset $\mathcal{D}$. Here, since h is determined by w, we write $\mathcal{L}(w, \mathcal{D})$. A popular choice for $\mathcal{L}$ is the mean squared error loss function:

$$\mathcal{L}(w, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}(h_w(x_i) - y_i)^2.$$

   $h_w(x)$ is typically denoted $\hat{y}$ and $|\hat{y} - y\,|$ is called a residual. Clearly we want to minimize this loss - a lower loss makes h closer in some sense to the actual y_i. Geometrically, it amounts to minimizing the sum of the squares of the vertical (in the y-dimension) distances from the hyperplane to each of the datapoints.

Why is the squared loss used? It is a differentiable and convex (and so has a global minimum). Why might we not want to use this loss? It is sensitive to outliers. For example, if we have a dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$ and we add a point (x_{n+1}, y_{n+1}) such that $y_{n+1} = 1000$, then the loss will increase significantly. A better loss in this case would be the so-called absolute loss, defined by

$$\mathcal{L}(w, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} |\hat{y}_i - y_i|.$$

Back to the problem. We use the training loss as our estimate of closeness, that is, the $\mathcal{D}$ in the loss is the training dataset.

Let us now write the loss in matrix form. Consider writing the dataset in matrix form:

$$X = \begin{pmatrix} \mathbf{x}_1^T \\ \mathbf{x}_2^T \\ \vdots \\ \mathbf{x}_n^T \end{pmatrix}_{n \times d}$$

Then

$$\mathbf{Xw} = \begin{pmatrix} \mathbf{x}_1^T\mathbf{w} \\ \mathbf{x}_2^T\mathbf{w} \\ \vdots \\ \mathbf{x}_n^T\mathbf{w} \end{pmatrix}_{n \times 1}$$

Also,

$$\mathbf{Y} = \begin{pmatrix} y_1 & y_2 & \dots & y_n \end{pmatrix}^T$$

so that

$$\mathcal{L}(w, \mathcal{D}) = \frac{1}{n}\|\mathbf{Xw} - \mathbf{Y}\|_2^2$$

where $|u|_2$ is the 2-norm of vector u. Thus $\mathcal{L}$ is also called an $\mathcal{L}^2$ loss. Also, for convenience, we will drop the boldface for vectors whenever clear from context. Thus

$$\mathcal{L} = \frac{1}{n}(Xw - Y)^T(Xw - Y)$$

2. *How do we find the best predictor? This is now an optimization problem.*

   Luckily enough, for this simple problem, there is a closed form solution. We can find the best predictor by solving the following optimization problem:

$$w_{\text{LS}} = \arg\min_{w \in \mathbb{R}^d} \mathcal{L}(w, \mathcal{D}).$$

   We can do this by setting the gradient of $\mathcal{L}$ wrt w to zero (we omit the $1/n$ because it does not affect the minimum). We have

$$\nabla_w \mathcal{L}(w, \mathcal{D}) = 2\mathbf{X}^\mathbf{T}(\mathbf{Xw} - \mathbf{Y})$$

so that

$$\nabla_w \mathcal{L}(w, \mathcal{D}) = 0 \iff X^T X w = X^T Y.$$

This is called the normal equation. It can be solved easily for w using row-reduction/standard linear solvers. If $\mathbf{X}^\mathbf{T}\mathbf{X}$ is invertible, then the unique solution is

$$\mathbf{w} = (\mathbf{X}^\mathbf{T}\mathbf{X})^{-1}\mathbf{X}^\mathbf{T}\mathbf{Y}.$$

Any such solution w is called the least squares solution.

2.2 The 1D case

Suppose that each x_i is a scalar. Then X is

$$X = \begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ \vdots & \vdots \\ 1 & x_n \end{pmatrix}$$

so that

$$X^T X = \sum_i X_i^T X_i = \begin{pmatrix} n & \sum_i x_i \\ \sum_i x_i & \sum_i x_i^2 \end{pmatrix} = n \begin{pmatrix} 1 & \overline{x} \\ \overline{x} & \overline{x^2} \end{pmatrix}$$

where X_i is the ith row of X and $\overline{t} = \frac{1}{n}\sum_i t_i$ is the sample mean of quantity $t = t_1, \dots, t_n$ over the dataset. $X^T X$ is invertible iff $\sigma_x^2 = \overline{x^2} - \overline{x}^2 \neq 0$. Suppose that is the case (it would be very uninteresting if the variance of the input was zero). Then

$$(X^T X)^{-1} = \frac{1}{n\sigma_x^2} \begin{pmatrix} \overline{x^2} & -\overline{x} \\ -\overline{x} & 1 \end{pmatrix}$$

and

$$X^T Y = \sum_i X_i^T Y_i = n \begin{pmatrix} \overline{y} \\ \overline{xy} \end{pmatrix}$$

So

$$\mathbf{w} = (\mathbf{X}^\mathbf{T}\mathbf{X})^{-1}\mathbf{X}^\mathbf{T}\mathbf{Y} = \frac{1}{\sigma_x^2} \begin{pmatrix} \overline{x^2} \cdot \overline{y} - \overline{x} \cdot \overline{xy} \\ \overline{xy} - \overline{x} \cdot \overline{y} \end{pmatrix}$$

So the intercept is

$$w_0 = \frac{\overline{x^2} \cdot \overline{y} - \overline{x} \cdot \overline{xy}}{\overline{x^2} - \overline{x}^2}$$

and the slope is

$$w_1 = \frac{\overline{xy} - \overline{x} \cdot \overline{y}}{\overline{x^2} - \overline{x}^2} = \frac{\overline{xy} - \overline{x} \cdot \overline{y}}{\overline{xx} - \overline{x} \cdot \overline{x}}$$

where the last expression is a little mnemonic for the formula. Notice the similarity of the slope to "y/x".

2.3 The general case

Suppose that X is $n \times d$ and $d > 1$. Then $X^T X$ is $d \times d$ and invertible iff $\mathrm{rank}(X) = d$ (why?). In this case, the least squares solution is

$$\mathbf{w} = (\mathbf{X}^{\mathbf{T}}\mathbf{X})^{-\mathbf{1}}\mathbf{X}^{\mathbf{T}}\mathbf{Y}$$

and the intercept is

$$w_0 = \frac{1}{n}\sum_{i=1}^{n} y_i - \sum_{j=1}^{d} w_j \overline{x_j}$$

and the slope is

$$w_j = \frac{\sum_{i=1}^{n} x_{ij} y_i - \overline{x_j}\sum_{i=1}^{n} y_i}{\sum_{i=1}^{n} x_{ij}^2 - \overline{x_j}\sum_{i=1}^{n} x_{ij}}$$

Chapter 3

Lecture 3

3.1 Converting nonlinear regression to linear regression

In the general regression case, h need not be linear. Consider the following simple case: $\mathcal{X} = \mathbb{R}$. Suppose, by looking at the datapoints plotted in $\mathbb{R}^2$ we notice that adding an x^2 term would probably help. That is, we want to consider hypotheses as

$$h_w = w_0 + w_1 x + w_2 x^2$$

The key point is that by defining $\phi : \mathbb{R} \to \mathbb{R}^3$ by $\phi(x) = (1, x, x^2)^T$, we can write $h_w(x) = w^T\phi(x)$. And then we can use the same linear regression algorithm to find the best w:

$$w = (\Phi^T\Phi)^{-1}\Phi^T Y$$

where

$$\Phi = \begin{pmatrix} \phi(x_1)^T \\ \phi(x_2)^T \\ \vdots \\ \phi(x_n)^T \end{pmatrix}$$

This is called a basis transformation. The reason for the name is that the elements of $\phi(x)$ form a basis for the set of hypotheses for datapoint x.

The general case: $\phi : \mathbb{R}^d \to \mathbb{R}^D$ where D is the dimension of the feature space. We typically write ϕ as

$$\phi(x) = \begin{pmatrix} \phi_1(x) \\ \phi_2(x) \\ \vdots \\ \phi_D(x) \end{pmatrix}$$

Then, we can write $h_w(x) = w^T\phi(x)$ and the loss function becomes

$$\mathcal{L}(w, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}(h_w(x_i) - y_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(w^T\phi(x_i) - y_i)^2.$$

As usual, the solution is then

$$w = (\Phi^T\Phi)^{-1}\Phi^T Y$$

where

$$\Phi = \begin{pmatrix} \phi(x_1)^T \\ \phi(x_2)^T \\ \vdots \\ \phi(x_n)^T \end{pmatrix}$$

Exercise. Is it useful to apply basis transformations to the label space?

3.1.1 Typical choices of basis transformations

1. Polynomial basis: $\phi(x) = (1, x, x^2, \dots, x^{D-1})^T$. This is the example we saw above.

2. Gaussian basis/Radial Basis Function: $\phi(x) = (1, e^{-\frac{|x-\mu_1|_2^2}{2\sigma_1^2}}, e^{-\frac{|x-\mu_2|_2^2}{2\sigma_2^2}}, \dots, e^{-\frac{|x-\mu_D|_2^2}{2\sigma_D^2}})^T$. This is a basis of D Gaussians with means $\mu_1, \dots, \mu_D \in \mathbb{R}^d$ and variances $\sigma_1^2, \dots, \sigma_D^2$.

3. Fourier basis: $\phi(x) = (1, \cos(\omega_1^T x), \sin(\omega_1^T x), \dots, \cos(\omega_D^T x), \sin(\omega_D^T x))^T$. This is a basis of D Fourier functions with frequencies $\omega_1, \dots, \omega_D \in \mathbb{R}^d$.

4. Piecewise Linear Basis.

5. Periodic Basis.

3.2 Digression: Some preliminaries

- Training Data. The dataset $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$.

- Training Error.

$$\mathcal{L}(w, \mathcal{D}) = \sum_{i=1}^{n} (h_w(x_i) - y_i)^2.$$

- Typically, the dataset is split into the Training set, Development Set and Testset. Typically, the split is 80-10-10. The training set is used to train the model, the development set is used to tune *hyperparameters* and the testset is used to evaluate the model. The testset is not used to tune the model in any way.

- The test error is a measure of the generalization of the model - better generalization is the holy grail of ML.

- How does one tune hyperparameters? Many ways: Grid Search, Random Search, Bayesian Optimization, etc.

3.2.1 Underfitting and Overfitting

Consider the following hypotheses class: all lines parallel to the axes: for example, $h_c(x) = c$, or $x_i = c$. Clearly, it cannot fit most data well - it is too simplistic and inflexible. This phenomenon is called underfitting, where the data is more complex than the hypothesis class.

The other end is more common: When the hypothesis class is far too general for the data. The model then fits the data almost perfectly (that is, reduces the loss to essentially zero) - and then it probably does not generalize as well to unseen points (why?). This situation is called overfitting.

Typically, one seeks to use such model complexity as minimizes the test error - see the picture below.

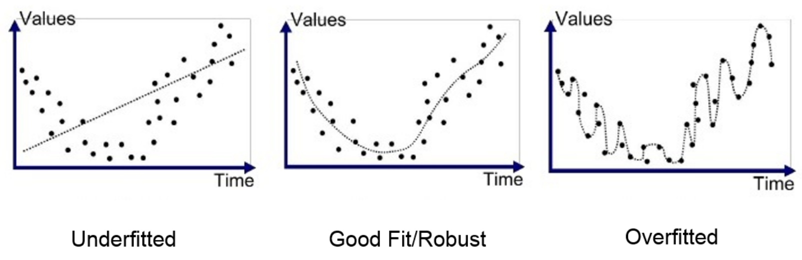


Figure 3.1: Different models for the same data. Taken from here.

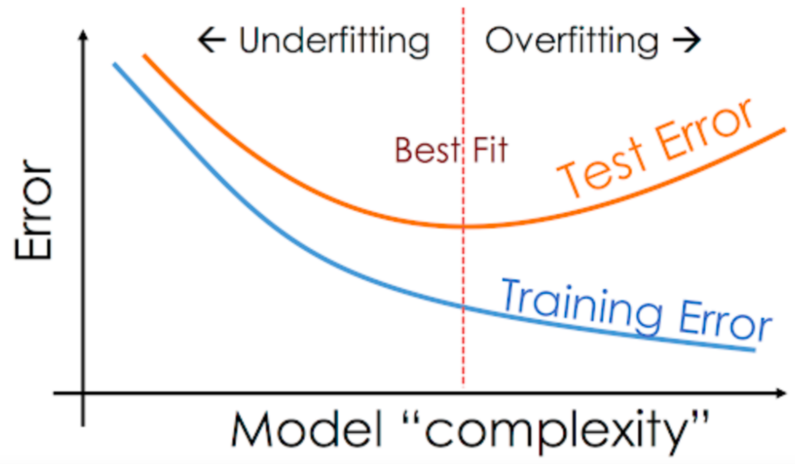


Figure 3.2: Underfitting and Overfitting. Taken from here.

Chapter 4

Lecture 4

4.1 Regularization

How does one combat overfitting? A coarse way is to play with the hyperparameters till we find what looks like the least test error. A more principled way is to use regularization.

Regularization is the following: we modify the loss function to explicitly penalize model complexity. The idea is that if we have two models with similar training error, we should prefer the simpler one. The loss now looks like so (where $\lambda \geq 0$ is a new hyperparameter):

$$\mathcal{L}(w, \mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n}(h_w(x_i) - y_i)^2 + \lambda R(w)$$

where $R(w)$ is the regularizer and is a measure of model complexity.

4.1.1 Penalty or shrinkage-based regularization

One way to regularize is to penalize the size of the weights. The idea is that if the weights are large, the variance across similar data points is more, and hence the model is more complex or sensitive. Oh, and typically, we do not penalize the bias term w_0, since that does not affect sensitivity - we describe this choice more formally when we discuss bias and variance.

A few common regularizers are:

- L2 regularization: $R(w) = ||w||_2^2$.
- L1 regularization: $R(w) = ||w||_1$.

4.2 Ridge Regression

Ridge regression is the name for linear regression with L2 regularization. The optimization problem is:

$$w* = \arg\min_w \frac{1}{n}|y - \Phi w|_2^2 + \lambda ||w||_2^2$$

We drop the $1/n$ term and scale λ by n for convenience. Let's differentiate:

$$0 = \nabla\mathcal{L}(w) = -2\Phi^T(y - \Phi w) + 2\lambda w$$
$$w^* = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T y$$

The key point is that when $\lambda > 0$, the matrix $\Phi^T\Phi + \lambda I$ is positive definite and hence invertible, yielding a unique minimum. This was actually the motivation for ridge regression, not the regularization that it also provided!

4.3 Lasso Regression

Lasso Regression uses the L2 loss with an L1 regularizer. Trivia: Lasso stands for Least Absolute Shrinkage and Selection Operator. The optimization problem is:

$$w* = \arg\min_{w} |y - \Phi w|_2^2 + \lambda ||w||_1$$

There is no closed form solution for this case. Ways around this:

- Quadratic Programming. This is a general technique for solving optimization problems of the form $\min_x \frac{1}{2}x^T Q x + c^T x$ subject to $Ax \leq b$. We can use this to solve the Lasso problem by setting $Q = \Phi^T \Phi$, $c = -2\Phi^T y$ and $A = -I, b = 0$. For the interested, look up QUBO.

- Iterative optimization algorithms. For example, Gradient Descent or Co-ordinate Descent.

4.3.1 Lasso vs Ridge

When to use L1/L2 as regularizer? Typically, L1 regularization yields sparser weight distributions (more zeros) as compared to L2 regularization. This is because the L1 regularizer is more aggressive in shrinking small weights.

Why? Consider the derivative of the regularizer: For L1, if the weight is $w > 0$, the derivative is 1 *whatever* the small value of w may be - and so, this needs to be removed (by setting w perfectly to 0) to minimize the loss. For L2, the derivative is $2w$ - and the contribution to the loss, for small enough w, goes unmodified. It is true that with enough iterations, the L2 regularizer will also shrink the weights to essentially 0, but it is obviously slower.

When is sparsity useful? When we want to do feature selection, i.e. when we want to find out which features are important and which are not, for example. Lasso is perhaps a good choice in this case.

Chapter 5

Lecture 5 - Gradient Descent

5.1 Gradient Descent

Gradient Descent or (GD) is an iterative optimization algorithm.

Suppose that we want to minimize a *differentiable* function $f : \mathbb{R}^n \to \mathbb{R}$. GD essentially uses the following fact from multivariable calculus:

Let the gradient ∇f be the function $\nabla f : \mathbb{R}^n \to \mathbb{R}^n$ be defined by

$$(\nabla f)(x) = \begin{pmatrix} \frac{\partial f}{\partial x_1} & \frac{\partial f}{\partial x_2} & \dots & \frac{\partial f}{\partial x_n} \end{pmatrix}^T$$

Then, one can choose α small enough such that $f(x - \alpha\nabla f(x)) < f(x)$, i.e. the function value decreases. This is because the gradient is the direction of steepest ascent, and the negative gradient is the direction of steepest descent. More particularly, from the Taylor expansion of f (we assume it is differentiable sufficiently many times) we have:

$$f(x - \alpha\nabla f(x)) = f(x) - \alpha\nabla f(x)^T\nabla f(x) + \frac{\alpha^2}{2}\nabla f(x)^T\nabla^2 f(x)\nabla f(x) + \dots$$

For α small enough so that $\frac{\alpha^2}{2}\nabla f(x)^T\nabla^2 f(x)\nabla f(x) + \dots$ is smaller in magnitude than $\alpha\,\|\nabla f(x)\|^2 > 0$ (such an α exists when we assume that the Hessian and higher derivatives are finite), we have $f(x - \alpha\nabla f(x)) < f(x)$.

The general GD algorithm is as follows:

```
w = w_0
for i = 1, 2, 3, ...
    w = w - alpha * grad(f)(w)
    if convergence condition is met
        break
    update_stepsize(alpha)
end
```

The most important bit,

$$w = w - \alpha\nabla f(w)$$

is called the gradient step or gradient descent step or gradient descent update step. It is imperative that we know exactly till when to do this (convergence) and exactly how to use it carefully (controlling the stepsize).

What is a possible convergence criterion? One possibility is to check if $\|\nabla f(w)\| < \epsilon$ for some small $\epsilon > 0$. Another possibility is to check if $\|w_{i+1} - w_i\| < \epsilon$ for small $\epsilon > 0$, where w_i is the weight vector after i iterations. A value ϵ used for convergence checks is called a tolerance.

The parameter α is called the stepsize or learning rate. How do we change α across iterations? Clearly, as the norm of the gradient decreases we would need α to decrease too for the gradient step to continue to reduce f.

A simple possibility is to use a constant stepsize α. Another possibility is to use a stepsize α that decreases with the number of iterations i - with the heuristic that with more iterations the gradient decreases in norm. For example, one can use $\alpha_i = 0.01/i$. In general, choosing the stepsize is a *black art* - there's a lot of work that's been done and there are many heuristics.

5.1.1 Gradient Descent for Linear Regression

Consider the standard regression loss without regularization. We have the gradient:

$$\nabla L(w) = \sum_{i=1}^{n} (w^T \phi(x_i) - y_i)(\phi(x_i)) = \Phi^T(\Phi w - y)$$

(upto a constant that can be subsumed into the stepsize). The gradient descent step is thus:

$$w = w - \alpha \sum_{i=1}^{n} (w^T x_i - y_i) x_i$$
$$w = w - \alpha \Phi^T (\Phi w - y)$$

5.1.2 Guarantees on convergence

5.2 Stochastic Gradient Descent

Consider the GD step for linear regression: it involves computing the gradient $\nabla L(w, \mathcal{D}_{train})$ which is a sum of the gradients over all the training examples. This is expensive when the number of training examples is large. SGD is a way to *approximate* the GD step with a single-sample gradient computation:

$$w = w - \alpha(w^T x_i - y_i) x_i$$

where (x_i, y_i) is a *random* training example. Under some conditions, one can show convergence to the optimal w even under SGD!

There is one obvious thing that nags us here: Is this the best way to perform GD for a large dataset? It is clearly extremely high variance, since of course the gradients for each sample are different. Is there a better way to approximate the GD step?

A popular choice in practice is to use a so-called minibatch of random datapoints from the dataset, and compute the gradient over this minibatch. This is called minibatch gradient descent or minibatch SGD. The minibatch size is typically a power of 2 such as $32, 64$. Its size is choisen based on the underlying hardware's parallelization capability.

5.3 The probabilistic view of Linear Regression

For training data $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$, we assume that the x_i's are target y_i's are generated by the following process:

$$y_i = f(x_i) + \epsilon_i$$

for some function f *in* our hypothesis class and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$. The ϵ_i's are independent (and of course, by definition, identically distributed) - that is, $\text{cov}(\epsilon_i, \epsilon_j) = 0$ for each $i \neq j$.

In the linear case, we have

$$y_i = w^T x_i + \epsilon_i$$

which means $y_i \sim \mathcal{N}(w^T x_i, \sigma^2)$. We thus have the likelihood

$$P(y_i|x_i, w) = \mathcal{N}(w^T x_i, \sigma^2)(y_i)$$

and the log-likelihood

$$\log P(y_i|x_i, w) = -\frac{1}{2\sigma^2}(y_i - w^T x_i)^2 - \frac{1}{2}\log(2\pi\sigma^2)$$

The likelihood of the entire dataset is

$$P(y_1, \ldots, y_n | x_1, \ldots, x_n, w)$$

We can simplify it as $P(y_2, \ldots, y_n|x_1, \ldots, x_n, w, y_1)P(y_1|x_1, \ldots, x_n, w)$. Next, $y_1|x_1, w$ is independent of $x_2, \ldots, x_n$ so the latter term is $P(y_1|x_1, w)$. The first term can be simplified:

$$P(y_3, \ldots, y_n|x_1, \ldots, x_n, w, y_1, y_2)P((y_2|x_2, w)|x_1, x_3, \ldots, x_n, y_1)$$

of which the latter term, since $y_2|x_2, w$ is independent of $x_1, x_3, \ldots, x_n$ and y_1, is $P(y_2|x_2, w)$. Continuing this way, we get

$$P(\mathcal{D}|w) \equiv P(y_1, \ldots, y_n|x_1, \ldots, x_n, w) = \prod_{i=1}^{n} P(y_i|x_i, w)$$

$$\log P(\mathcal{D}|w) = -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(y_i - w^T x_i)^2 - \frac{n}{2}\log(2\pi\sigma^2)$$

which is the same as the loss function we used for linear regression! (upto a constant). This means that minimizing the loss function is the same as maximizing the log-likelihood of the data under the model - that is, the maximum likelihood estimate of w is the same as the least squares estimate of w.

5.3.1 A major limitation of Maximum Likelihood Estimation

Since MLE maximizes to the data - it is perfectly training-data driven - it is prone to overfitting, especially when the amount of data is small and not fully representative of the true distribution.

An extreme example would be the following: Suppose that our data is a string of 0s and 1s (so x_is are not present, or can be treated as a temporal element) representing the outcomes of a biased coin tossed repeatedly. The maximum likelihood estimate for the bias is the average of the data. This can be a terrible estimate of the bias if the number of tosses is small.

To fix this overdependence on data, we try another technique - Bayesian estimation, where the idea of using “previous domain knowledge” can be naturally incorporated, often leading to a slightly lesser dependence on raw data.

Chapter 6

Lecture 6

6.1 Bayesian Parameter Estimation

In typical ML, one seeks to find values for parameters of the model that best fit the data. In the Bayesian framework, each parameter is a random variable itself, with some distribution. The distributions of the parameters vary according to Bayes' rule, as data is observed. Finally, we can set the value of a parameter using a maximum/expectation etc over its final distribution. This is called the "Bayesian" idea in contrast with MLE which is a "frequentist" idea.

6.1.1 Bayes' Rule

Let θ be the set of parameters (we shall just call it the parameter). The initial distribution that one has on θ is called the prior $P(\theta)$. After observing data $\mathcal{D}$, the distribution on θ is called the posterior $P(\theta|\mathcal{D})$. Bayes' rule tells us how to compute the posterior from the prior and the likelihood $P(\mathcal{D}|\theta)$:

$$P(\theta|\mathcal{D}) = \frac{P(\mathcal{D}|\theta)P(\theta)}{P(\mathcal{D})}$$

The P here is the joint distribution of the data and the parameter - defined by $P(\mathcal{D}, \theta) := P(\mathcal{D}|\theta)P(\theta)$. The likelihood $P(\mathcal{D}|\theta)$ is the same as before -

$$P(\mathcal{D}|\theta) = \prod_i P((x_i, y_i)|\theta)$$

We typically use the following more convenient expression:

$$P(\theta|\mathcal{D}) \propto P(\theta)P(\mathcal{D}|\theta)$$

$$\text{posterior} \propto \text{prior} \times \text{likelihood}$$

where there is proportionality because the denominator is independent of θ. We shall see why this is more convenient in just a little while. Next, what do we do with the posterior distribution?

6.1.2 Bayesian Estimation: The idea

In Bayesian estimation, we typically assume that the posterior is a good approximation to the true distribution of the parameter. Why is this a good idea? Consider the following extreme case - the prior is *perfect*. That is, if the data was generated with parameter θ_0, then $P(\theta) = 0$ at all other values of θ. Then, the posterior is also perfect - it is 0 everywhere except at θ_0! Now suppose that the prior peaks at θ_0 but is not constant. By the MLE theorem, the likelihood is maximized at θ_0 which means that the peak at θ_0 in the posterior is actually higher. And when we now use this as the prior and add in the effect of even more data, we get an even higher peak and so on - the posterior approaches the true distribution of the parameter θ.

Suppose now that the prior peaks somewhere else but has a nonzero probability at θ_0. Then, the posterior will have a larger probability at θ_0 but it may still not be the peak. This is because the likelihood is maximized at θ_0 but the prior is not - the posterior is a compromise between the prior and the likelihood (prior knowledge is respected to some extent). Of course, with more and more data, indeed, the peak at θ_0 begins to form, and our distribution gets better - obeying the general principle that more data is better.

Finally, the other extreme case: the prior has zero probability density at θ_0. In this case, every posterior will have zero probability density at θ_0 - no amount of data will change this. (This is the Bayesian way of saying that the parameter θ_0 is not in the hypothesis class.) And in this case, yes, the posterior will be suboptimal and will not reflect the true parameter.

Thus, a good prior is required. A better prior will require a smaller amount of data to get a good posterior. A bad prior will require a larger amount of data to get a good posterior. In the extreme case of a perfect prior, no data is required to get a good posterior. In the case of a perfectly incorrect prior, no amount of data will get the perfect posterior.

6.2 Maximum a Posteriori (MAP) estimation

Suppose that we have used our dataset $\mathcal{D}$ to construct a posterior. How do we pull out a value for the parameter to use from this posterior. A natural way, as we have thus far seen, is to use the mode (the peak) of the posterior. This is called the maximum a posteriori (MAP) estimate of the parameter. That is, the MAP estimate is:

$$\theta_{MAP} := \arg\max_{\theta} P(\theta|\mathcal{D})$$

6.2.1 Conjugate Priors

Chapter 7

Lecture 7

7.1 Bias and Variance of estimates

We now take a step ahead. Suppose we now have an estimator for some parameter. How "good" is an estimator? Well, we can easily find a measure for this - the mean squared error (MSE) of the estimator. Suppose that the true value of the parameter is θ_0. Then, the MSE of an estimator $\hat{\theta}$ is defined as:

$$\mathrm{MSE}(\hat{\theta}) := \mathbb{E}\left[(\hat{\theta} - \theta_0)^2\right]$$

where the distribution is over the true distribution $P_{\mathcal{D}}$ of the dataset $\mathcal{D}$ and the randomness used to compute the estimator $\hat{\theta}$. The MSE is a measure for how well the estimator will do on unseen test data (which is more or less all we care about). The test loss on (a lot of) unseen data when using estimator $\hat{\theta}$ is given by

$$\mathcal{L}_{\text{test}} = \frac{1}{N_{\text{test}}} \sum_{i=1}^{N_{\text{test}}} (\hat{\theta} - \theta_0)^2$$

where the sum is over all test examples. The MSE is the expected value of this loss, and by the law of large numbers, for large N_{test}, the test loss will be close to the MSE. So, the MSE is a good measure for generalizability.

7.2 The true definition of a regression problem - Statistical Learning Theory

Let us place ourselves in the world of random variables and probability spaces. Let $X \in \mathbb{R}^p$ denote a real valued random input vector, and $Y \in \mathbb{R}$ a real valued random output variable, with joint distribution $p(X, Y)$. We seek a function $f(X)$ for predicting Y *given* values of the input X. This theory requires a loss function $L(Y, f(X))$ for penalizing errors in prediction, and by far the most common and convenient is squared error loss: $L(Y, f(X)) = (Y - f(X))^2$. This leads us to the expected prediction error (EPE) or expected test error criterion for choosing f:

$$\mathrm{EPE}(f) = \mathbb{E}\left[(Y - f(X))^2\right]$$

where the expectation is taken over the joint distribution of X and Y. The solution f to the above minimizes the expected test error, and is called the regression function. One can 'factorize' the EPE with conditioning:

$$\mathrm{EPE}(f) = \mathbb{E}_X\left[\mathbb{E}_{Y|X}\left[(Y - f(X))^2\right]\right]$$

The inner expectation is over the conditional distribution of Y given X, and is a function of X. We sometimes write $\mathbb{E}_{Y|X}[\cdot]$ as $\mathbb{E}[\cdot|X]$ for brevity. It suffices to minimize the conditional expectation pointwise: at each x,

$$f(x) = \arg\min_{c\in\mathbb{R}} \mathbb{E}_{Y|X=x}\left[(Y-c)^2\right] = \mathbb{E}[Y|X=x]$$

The function $f(x) = \mathbb{E}[Y|X=x]$ is called the regression function or the regression curve. It is the best predictor for Y. This means that the EPE is *always* at least $\mathbb{E}_X\left[\mathbb{E}\left[(Y-\mathbb{E}[Y|X])^2|X\right]\right] = \mathbb{E}_X[\mathrm{var}(Y|X)] \geq 0$. This is zero iff Y is a deterministic function of X, i.e. $Y = f(X)$ for some function f. In this case, the regression function is f itself. This minimum error is called the irrecoverable error or irrecoverable noise.

The nearest neighbors method is a non-parametric method for directly estimating the regression function. See ESL for more details.

Linear regression is

7.2.1 The bias-variance decomposition

Suppose that the true variance is θ_0. Then, we can write the MSE as follows: and the MSE is

$$\mathrm{MSE}(f) = \mathbb{E}\left[(f(x)-\theta_0)^2\right]$$

where the expectation is over the randomness in the dataset. We can decompose the MSE into two parts: the bias and the variance of the estimator. The bias is the error due to the estimator not being able to capture the true value of the parameter, and the variance is the error due to the estimator being too sensitive to the randomness in the dataset. The bias-variance decomposition is as follows:

$$\begin{aligned}
\mathrm{MSE}(\hat\theta) &= \mathbb{E}_{P_{\mathcal{D}},\epsilon}\left[(\hat\theta - \mathbb{E}[\hat\theta] + \mathbb{E}[\hat\theta] - \theta_0)^2\right] \\
&= \mathbb{E}\left[(\hat\theta-\mathbb{E}[\hat\theta])^2\right] + \mathbb{E}\left[(\mathbb{E}[\hat\theta]-\theta_0)^2\right] + 2\cancelto{0}{\mathbb{E}\left[(\hat\theta-\mathbb{E}[\hat\theta])\right]}(\mathbb{E}[\hat\theta]-\theta_0) \\
&= \mathrm{var}(\hat\theta) + \mathrm{bias}(\hat\theta)^2
\end{aligned}$$

7.3 How does regularization reduce variance?

Chapter 8

Logistic Regression

We now move to (binary) classification. We would like a function $f : \mathcal{X} \to \mathcal{Y} = \{0, 1\}$; that is, find the best deterministic estimate to a probability distribution over $\mathcal{X} \times \{0, 1\}$. What loss to choose? A natural choice is the discrete loss, determined by a matrix L: L_{ij} is the loss incurred when predicting j when the true label is i. For example, if $L_{ij} = 1$ for $i \neq j$ and $L_{ii} = 0$, then we have the 0-1 loss. If $L_{ij} = |i - j|$, then we have the absolute loss. If $L_{ij} = (i - j)^2$, then we have the squared loss. Only i and j are discrete, and in the binary classification case, the loss matrix is 2×2. Typically one uses the 0-1 loss - $L(i, j) = \mathbb{1}_{[i \neq j]}$. The EPE has the form:

$$\text{EPE}(f) = \mathbb{E}_X \left[\mathbb{E}_{Y|X} [L(Y, f(X))]\right]$$

Minimizing pointwise, we have the regression function:

$$f(x) = \arg\min_{y \in \mathcal{Y}} \mathbb{E}_{Y|X=x} [L(Y, y)] = \arg\min_{y \in \mathcal{Y}} \mathbb{E}_{Y|X=x} \left[\mathbb{1}_{[Y \neq y]}\right] = \arg\max_{y \in \mathcal{Y}} P(Y = y|X = x)$$

which is comforting - we classify based on the most likely label. This is called the Bayes classifier.

In our case, we have $\mathcal{Y} = \{0, 1\}$, so we have $P(Y = 1|X = x) = 1 - P(Y = 0|X = x)$. So we have $f(x) = \mathbb{1}_{[P(Y=1|X=x)>0.5]}$.

8.1 Logistic Regression

The k-NN classifier estimates the probabilities by counting the number of points in each class in a neighborhood - only the averaging turns into a majority vote instead.

The linear model is next, of course. We try to approximate the probability distribution $P(Y = 1|X = x)$ with a linear function $\beta^T x$. Classification is then done as

$$f(x) = \begin{cases} 1 & \text{if } \beta^T x > 0.5 \\ 0 & \text{if } \beta^T x \leq 0.5 \end{cases}$$

There is one problem that jumps up here - $\beta^T x$ may be negative, and then we would not want to trust it to be a probability. How do we ensure values lie in $[0, 1]$ - in comes the sigmoid function.

The sigmoid function is

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

The nice thing about the sigmoid is that its range is precisely $(0, 1)$ for $x \in \mathbb{R}$. We replace $\beta^T x$ by $\sigma(\beta^T x)$. Our classifier is now:

$$f(x) = \begin{cases} 1 & \text{if } \sigma(\beta^T x) > 0.5 \\ 0 & \text{if } \sigma(\beta^T x) \leq 0.5 \end{cases}$$

This is called logistic regression. (why logistic?: The sigmoid function was already used in Physics with the name logistic function, see here) What is the loss? Perhaps the negative log-likelihood?

8.1.1 ML estimation for β

We have the loss for dataset $\mathcal{D} = \{(x_i, y_i)\}$ -

$$\mathcal{L} = \log \prod_i P(y_i | x_i, \beta)$$

Notice that we can write

$$\begin{aligned} \log P(y_i|x_i, \beta) &= y_i \log P(y_i = 1|x_i, \beta) + (1 - y_i) \log(1 - P(y_i = 1|x_i, \beta)) \\ &= y_i \log f(x) + (1 - y_i) \log(1 - f(x)) \end{aligned}$$

So the ML estimate is

$$\beta_{ML} = \arg\min - \sum_i (y_i \log f(x_i) + (1 - y_i) \log(1 - f(x_i)))$$

The loss

$$\mathcal{L} = - \sum_i (y_i \log f(x_i) + (1 - y_i) \log(1 - f(x_i)))$$

This is the loss that we use and is called the binary cross-entropy loss, since it represents the cross-entropy between the distributions.

Consider the logistic regression functino $f(x)$. The loss is

$$\mathcal{L} = - \sum_i (y_i \log \sigma(\beta^T x_i) + (1 - y_i) \log(1 - \sigma(\beta^T x_i)))$$

and the gradient of the loss can be worked out to be

$$\nabla_\beta \mathcal{L} = \sum_i (\sigma(\beta^T x_i) - y_i) x_i$$

There is no closed form solution for β; typically gradient descent is used. The cross-entry loss is convex, so gradient descent typically converges nicely to the global minimum.

8.1.2 The decision boundary

The decision boundary is the set of points where the probability of being in each of the two classes is equal, that is, $f(x) = 0.5 \implies \beta^T x = 0$. So the decision boundary is a hyperplane through the origin, with normal vector β.

8.1.3 Optimizations

As with regression, one can use basis functions and regularization. The basis functions are used to find nonlinear decision boundaries, and regularization is used to prevent overfitting.

Since the decision boundary passes through the origin, **scaling β does not change the classification**. Let's look at this a little more carefully.

Consider a point x close to the decision boundary. Then $\beta^T x$ is close to 0. So we have

$$\sigma(\beta^T x) \approx \frac{1}{2} + \frac{\beta^T x}{4}.$$

Scaling β will not change the classification, but will keep increasing the likelihood of x being in the class it was alloted. Consider another point far from the decision boundary, so $f(x) \approx 1$ (or 0). A larger β is less sensitive to the difference between each a near (the decision boundary) and far point. Is this good? It depends on the data. If the data is well separated, then this is good. If the data is not well separated, then this is bad.

Let us compute the variance of the estimator:

Chapter 9

Decision Tree Classifiers

We have an importnat question now: which features are important? This is called the model interpretability problem. This is extremely useful, especially when the feature dimension is large. Today we will look at decision trees (for classification), which are interpretable models.

9.1 Decision trees

Decision Trees are interpretable models where the final prediction is a disjunction of conjunctions of features. For example, we might have a decision tree that looks like this:

Decision trees explicitly separate data by means of a tree. Each node is a feature, and each edge is a value of the feature. The leaves are the predictions. The path from the root to a leaf is a conjunction of features, and the final prediction is a disjunction of these conjunctions.

9.1.1 Learning decision trees

How do we learn decision trees? We need to learn the tree structure, and the prediction at each leaf. Finding the smallest optimal decision tree with respect to some metric on trees is NP-hard. Heuristics are the way to go.

We will use a simple greedy approach to build the tree. We use the information gain as the metric to choose the best feature at each step. In other words, we choose a variable at each step that best splits the set of items. It is quite like 3b1b's Wordle video.

A simple template for building DTs is as follows:

Proposition 9.1. *Building a decision tree.*

1. *Start with all data at the root node.*
2. *Choose the best feature to split the data.*
3. *Split the data into two sets based on the feature.*
4. *Recurse on the two sets.*

A few obvious questions: What do you mean by best feature, and how do you compute it? And then, how do you split the data? And finally, what is a good stopping criterion / basecase?

9.2 How to split the data

Consider the following two cases (the training dataset has 13 elements, 6 in class $+$, and 7 in class $-$):

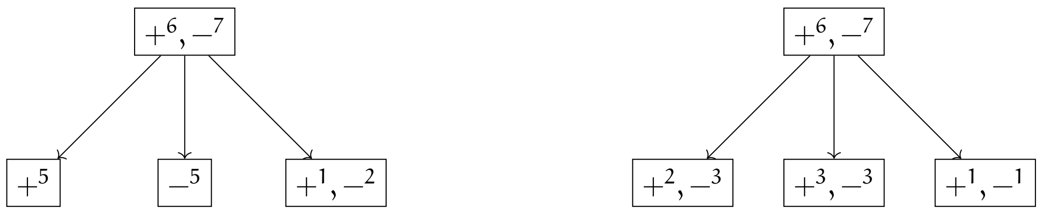


Figure 9.1: Two different ways to split the data. Which one is a better split?

Intuitively, the first split has done a lot of separation already, and so we expect the recursion is trivial - we don't even need to recurse on the first two nodes. The second split, on the other hand, has not done much separation yet, and perhaps the final tree from hereon is longer. So we would like to choose the first split. How do we formalize this intuition?

A note, though: The second split *may* lead to a shorter final tree, if good attributes can be chosen in the second step; so greedy is not guaranteed to be optimal. This is a taste - the fact that shortenings of the tree are not local - of why this problem is NP-hard.

The questions remain: how do we split the data well? Which attributes to consider? We go with the greedy principle of always picking up the attribute that separates the data best. Onto defining a notion for separation. We would like that the "separation" is more in the first split above, and less in the second. A simple measure of separation exists, and is called the information gain.

9.2.1 Information Gain

The information gain is a measure of how much information a feature gives us about the class. It is defined as follows:

Definition 9.2 (Information Gain). The information gain of a feature X is defined as

$$IG(X) = H(Y) - H(Y|X)$$

where $H(Y)$ is the entropy of the class, and $H(Y|X)$ is the entropy of the class given feature X.

The entropy of a random variable X is $\sum_x -p_x \log p_x$ where $p_x = p(X = x)$. The entropy of a random variable X is a measure of how much information is gained when we know the value of X. The entropy is maximized when X is uniformly distributed, and is minimized when X is a point mass.

For a dataset $\mathcal{D}$, the entropy of the class is $H(Y) = -\sum_y p(y) \log p(y)$ where $p(y)$ is the fraction of elements in $\mathcal{D}$ with label y. The entropy of the class given feature X is $H(Y|X) = \sum_x p(x)H(Y|X = x)$ where $p(x)$ is the fraction of elements in $\mathcal{D}$ with feature value x, and $H(Y|X = x)$ is the entropy of the class that is given feature value x.

The conditional entropy $H(Y|X)$ for random variables X, Y is defined to be $\sum_{x,y} p(x, y) \log p(y|x)$. One can rewrite the conditional entropy as $\sum_x p(x)H(Y|X = x)$, which is the expected value of the entropy of Y given X. If X largely determines Y - a good separator - then $H(Y|X)$ is small, and so $IG(X)$ is large.

The best separator is defined then to be

$$x^* = \arg\max_x IG(X) = \arg\min_x H(Y|X = x)$$

Alright. So that's done. Before we finish, let us note a few more measures of separation:

- The Gini index: $1 - \sum_i p_i^2$ where p_i is the fraction of label i in the data.
- Just the entropy of the label. What does this mean?

9.2.2 Splitting on numerical data

What if a feature is a continuous scalar? We need to discretize it. We can do this by sorting the data, and then choosing the split that gives the best information gain. Note that we choose splits only from the set of midpoints of adjacent values of data whole labels differ. Why does this work? In a certain sense, clusters form at similar attribute values, so this may not be a bad thing to try. As usual, finding the best split from the available splits is done by information gain.

9.3 When do we stop?

As we continue to recurse, the size of the dataset that we run the feature-choosing algorithm with (the separator-choosing algorithm) is decreasing. The approximation of summations over the dataset to expectations is getting worse - essentially, we come close to considering even irrelevent features that may not reflect actual distribution in the testset - and move to overfit. Clearly, we need to stop in this case.

Some stopping criteria:

1. Stop when the data is pure - all elements in the dataset are of the same label.
2. Stop if the number of instances at a node is less than some fixed k.
3. Stop if the maximum gain at a mode is below some threshold - there is not much to separate on, we might as well keep the data together. Used in conjunction with: stop if the total number of leaves exceeds some threshold.

There is an alternative to stopping criteria: pruning. We cover one variant, called reduced error pruning. The idea is to build the tree till the end, and then "prune" the tree by removing nodes that do not improve the error on the validation set. Pruning a node means removing the subtree rooted at that node, and replacing it with a leaf node with the majority label in the subtree. At each round, we remove the node that gives the best improvement in error on the validation set. This is a form of regularization.

A wonderful website explaining decision trees is here.

9.4 Bootstrap aggregation or BAGGing

Consider a dataset $\mathcal{D}$ of size n. *Create* M datasets of same size n from $\mathcal{D}$ by sampling *with* replacement. This is called bootstrapping - this reduces variance (and heya, that is just like bootstrapping in reinforcement learning). Now, train M models on these datasets. The final prediction is the majority vote over the M models. This is called bootstrap aggregation.

9.5 Random Forests

Random forests are a special case of bagging - they are bagged decision trees (hence the name "forest"), with a twist: for each DT we also randomly select a subset of features to split the DTs on. This is done to reduce correlation between the models - if there was a dominant feature, then all the models would separate on this, and there would not be much point in many models. But you ask, am I not using (possibly) irrelevant features in some of the decision trees? Well, yes. But the majority vote will take care of that. The tiny bit of uncertainty that still lingers serves as a counter to overfitting.

Chapter 10

The Perceptron

10.1 The Perceptron for binary classification

A linear model to separate the training data (into two classes). Learns vector w so that the data is best separated using the hyperplane $x^T x = 0$. At test time, the test instance x is given the label $h_w(x) = \text{sgn}(w^T x)$ - thus the classes are $y = -1, 1$. A couple of points to note: the perceptron learns online - it can adapt to more instances being added at any time - and it is error-driven - it only updates when it makes a mistake.

10.1.1 The algorithm

Algorithm 1 Perceptron

```
1: $w \leftarrow 0$
2: while not converged do
3:     for $(x, y) \in \mathcal{D}$ do
4:         if $y \neq \text{sgn}(w^T x)$ then
5:             $w \leftarrow w + yx$
6:         end if
7:     end for
8: end while
```

In particular, the update is $w \leftarrow w + x$ if $y = +1$, and $w \leftarrow w - x$ if $y = -1$. Is the perceptron guaranteed to learn on each mistake? Consider an x with $y \neq \text{sgn}(w^T x)$. We set $w' = w + yx$ so $w'^T x = w^T x + y \|x\|^2$, or $yw'^T x = yw^T x + y^2 \|x\|^2$. Since $y^2 = 1$, we have $yw'^T x = yw^T x + \|x\|^2 > yw^T x$. The perceptron classifies x correctly iff $yw^T x > 0$. So indeed, if it was negative initially, it does move up by $\|x\|^2$ - closer to the correct prediction.

Caveat. The new weight vector classifies the mistake x better, but it might make new mistakes on other instances. Wonderfully, though, the algorithm will still work out.

10.1.2 Convergence

The perceptron is guaranteed to converge in a finite number of iterations if the data is linearly separable.

10.1.3 The hinge loss

Can we look at the perceptron as minimizing a loss function? If an instance is correctly classified $\iff yw^\top x > 0$, we should add 0 to this loss. Otherwise, we should penalize on something so that $w \leftarrow w + yx$ is the right gradient update. $\nabla_w \mathcal{L} = -yx \implies \mathcal{L} = -yw^\top x$. And if the instance is correctly classified, we add 0. This gives us the loss $\max(0, -yw^\top x)$ for instance x. Summing up we get the loss that the perceptron algorithm minimizes:

$$\mathcal{L}_{\text{hinge}}(w) = \sum_{(x,y)\in\mathcal{D}} \max(0, -yw^\top x)$$

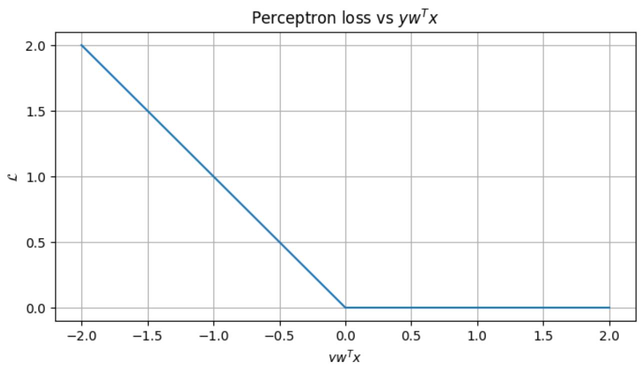


Figure 10.1: The hinge loss

This probably means that we can use trusty SGD on this perceptron loss. Except...it's not quite differentiable. Here's where the cool idea of a subgradient kicks in. The subgradient of a convex function f at a point x is a vector g such that $f(y) \geq f(x) + g^\top(y - x)$ for all y. If f is differentiable at x, the subgradient is forced to be the gradient. The neat thing about the subgradient is that we can substitute it for the gradient in gradient descent, and it will still work.

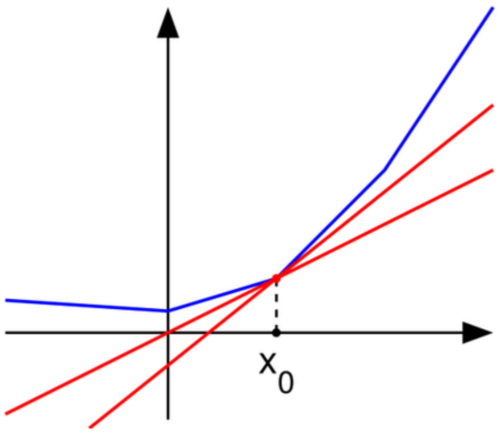


Figure 10.2: The subgradient of a convex function at a point x_0 is a vector g with f lying above the curve through x_0 with slope g. Taken from Wikipedia.

For the hinge loss, a valid subgradient is $-yx$ if $yw^\top x < 1$, and 0 otherwise.

10.2 Convergence bounds of the Perceptron algorithm

Consider a linearly separable dataset $\mathcal{D}$, i.e. one such that there exists a (unit) vector u such that $Y = \text{sgn}(u^T x)$ for every $x, y \in \mathcal{D}$. Let us suppose without loss of generality that each $x_i \in \mathcal{D}$ has norm atmost 1.

Define the margin of separation of $\mathcal{D}$ w.r.t u be defined as

$$\gamma = \min_{x \in \mathcal{D}} |u^T x|$$

that is, γ is the minimum distance of a point $x \in \mathcal{D}$ from the hyperplame $u^T x = 0$.

Theorem 10.1 (Navikoff, '63). *If there exists a unit vector u such that for every $(x, y) \in \mathcal{D}$, we have $y u^T x \geq \gamma$. Then the Perceptron algorithm will make no more than $\frac{1}{\gamma^2}$ mistakes on the training dataset when initialized with $w = 0$.*

Proof. We prove the following: w appears to converge toward the optimal (read: zero loss) vector u. Consider the quantity $w^T u$. After an update, we set $w_{i+1} = w_i + yx$, which gives

$$w'^T u = w^T u + y x^T u \geq w^T u + \gamma$$

By induction, after k steps, we have $w_k^T u \geq k\gamma$. Could the dot product increase due to the norm of w increasing, and not necessarily because we are getting closer to u? No! For each i, we have

$$\|w_{i+1}\|^2 = \|w_i\|^2 + 2y w^T x + \|x\|^2 \leq \|w_i\|^2 + 0 + 1 = \|w_i\|^2 + 1$$

which means that $\|w_k\| \leq \sqrt{k}$ for every k. If the Perceptron algorithm makes k mistakes (there were k updates of the weight vector) then

$$\sqrt{k} \geq \|w_k\| \geq w_k^T u \geq k\gamma \implies k \leq \frac{1}{\gamma^2}$$

That means, there *cannot* be a $\left(1/\gamma^2 + 1\right)$th mistake - there will be at most $1/\gamma^2$ mistakes. $\square$

If the data is linearly separable, as we assumed, then the premises of the above theorem hold, thus: The perceptron needs atmost $1/\gamma^2$ rounds of the training dataset to classify it perfectly, where γ is the maximum margin of separation over all linear separators for $\mathcal{D}$.

If the previous proof wasn't convincing enough, we can also supply a direct proof based on the fact that an update always reduces the loss by a substantial amount.

Proof. (direct, from the loss) We again assume that the norm of $x \in \mathcal{D}$ is bounded above by 1. On an update, we have:

$$y w_{i+1}^T x = y w_i^T x + \|x\|^2 \leq y w_i^T x + 1$$
$$\max(0, -y w_{i+1}^T x) \leq \max(0, -y w_i^T x) - 1$$

Hmm, the issue is that this is not exactly the loss - the case $w^T x = 0$ is a mismatch whatever is y. We need to make the loss $\max(0, \epsilon - y w^T x)$ or similar. $\square$

Chapter 11

Support Vector Machines

11.1 Motivation

It is desirable to maximize the "margin", i.e. the distance of the closest point in either class from the hyperplane should be maximized while correctly predicting all the training points. Support Vector Machines are a way to do this (and that's why they are also called 'Maximum-margin classifiers').

11.2 The Hard-Margin SVM

A hyperplane separator/decision boundary is a set of points $w^{\mathsf{T}}x + b = 0$. The sign of $w^{\mathsf{T}}x + b$ determines the class of x. The distance of a point x from the hyperplane is given by $\frac{w^{\mathsf{T}}x+b}{\|w\|}$. The margin is the distance of the closest point in either class from the hyperplane. The hard-margin SVM is the problem of finding the hyperplane with the maximum margin. Notice that one can actually consider finding *two* hyperplanes, one on either side of the decision boundary with the margin being the distance between these two hyperplanes.

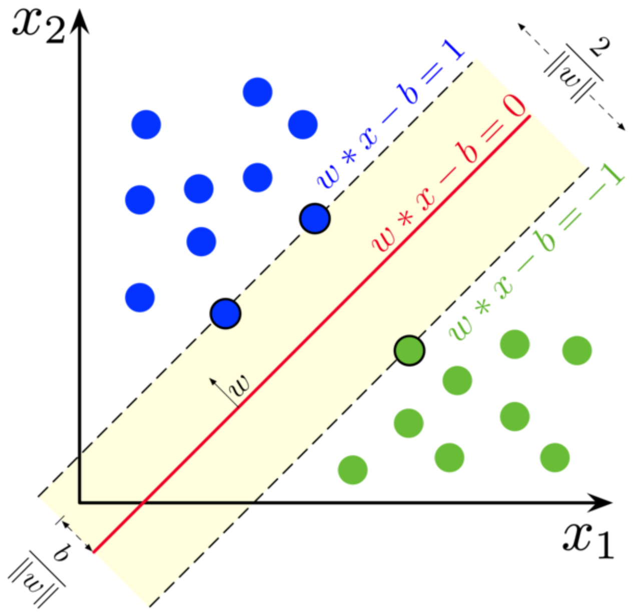


Figure 11.1: The hard-margin SVM

Suppose the two hyperplanes at the margin are $w^{\mathsf{T}}x + b = \pm c$, $c \neq \mathbf{0}$. By renormalizing w if needed, we may assume $c = 1$. Then the margin is given by $\frac{2}{\|w\|}$. So the problem is equivalent to maximizing $\|w\|$ subject to the constraints

11.3 The SVM Optimization Problem

Definition 11.1 (General CSP). The general constraint satisfaction problem is defined as follows:

$$\begin{aligned} \text{compute } \min_{W} f(w) & \\ \text{subject to } g_1(w) &\leq 0 \\ g_2(w) &\leq 0 \\ &\vdots \\ g_n(w) &\leq 0 \end{aligned}$$

One typically also calls this the primal problem. Any value of $w \in W$ that satisfies the constrains is called a feasible solution.

The SVM optimization problem is a special case of the general CSP. The objective function is the margin, and the constraints are that the points are classified correctly.

The lagrangian for a CSP is defined like so:

$$L(w, \alpha) := f(w) + \sum_{i=1}^{n} \alpha_i g_i(w)$$

An equivalent formulation of the primal is to minimize the maximum of the lagrangian: Compute $\min_w \max_{\alpha \geq 0} L(w, \alpha)$. Why the equivalence? For every feasible w, it is trivial to see that $\max_{\alpha \geq 0} L(w, \alpha) = f(w)$. Conversely, if w is not feasible, then $\exists i$ such that $g_i(w) > 0$. In this case, $\max_{\alpha \geq 0} L(w, \alpha) = \infty$. Thus the minimum of the maximum is $\min f(w)$ (or ∞ if no feasible w).

The dual problem is defined by

$$d^* = \max_{\alpha \geq 0} \min_{W} L(w, \alpha)$$

It is clear that $d^* \leq p^*$ (the max of a min is at most the min of a max - notice the connection to the true formula $\exists x \forall y f(x, y) \implies \forall y \exists x f(x, y)$). This is called weak duality. Remarkably, it turns out that (under some conditions - The Karush-Kuhn-Tucker conditions) $d^* = p^*$ for convex problems. This is called the strong duality theorem.

The SVM problem satisfies these constraints, and so it suffices to solve the dual problem (which could be easier because there are no constraints to be satisfied)

11.4 The SVM Dual Problem

Proposition 11.2. *The dual problem for SVM is:*

$$\max_{\alpha_i \geq 0} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i^T x_j$$

such that $\sum_i \alpha_i y_i = 0$. The weight vector is given by $w = \sum_i \alpha_i y_i x_i$.

In the dual form of the soft-margin SVM, only the constraints on the α_i change (to $0 \le \alpha_i \le c$). The objective function remains the same.

A few observations on the dual form:

- If the solution to the primal problem if w^* and the solution to the dual problem is α^*, then the KKT conditions on w^*, α^* hold for the SVM problem. One of them, called the complementarity constraint, is of most interest:

$$\alpha_i^* g_i(w^*) = 0 \ \forall i$$

  So $\alpha_i > 0$ corresponds to those training points for which $g_i(w^*) = 0$, or $y_i({w^*}^T x_i + b^*) = 0$. These are exactly the support vectors!

- The dual form operates on *inner* products of training instance pairs. In fact, even during predictions, we take inner products:

$$\hat{y} = \text{sgn}({w^*}^T x + b^*) = \text{sgn}\left(\sum_i \alpha_i^* y_i (x_i^T x) + b^*\right)$$

  This will be crucial when we talk about nonlinear boundaries and the kernel trick.

11.5 The Kernel Trick

Moving beyong hyperplane separators - onto non-linear decision boundaries. The typical idea: replace data points x_i with feature vectors $\phi(x_i)$. The decision boundary is now a hyperplane in the feature space, but nonlinear in the original input space.

We have two questions:

1. How do we find the feature vectors $\phi(x_i)$? The quintessential problem.
2. The feature space could be very high-dimensional, and so the computation of the inner product $\phi(x_i)^T \phi(x_j)$ could be very expensive.

Define the kernel function $K : \mathbb{R}^n \to \mathbb{R}^n$ by $K(x, y) = \phi(x)^T \phi(y)$. Suppose that K was efficiently computable. Then, using the observation that everything (training and prediction) is done using inner products, we can replace all inner products with K and be able to compute it fast. This is the kernel trick.

Consider the following example for instances $x = (x_1, x_2)$ and $y = (y_1, y_2)$ and the feature map $\phi(x) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)^T$. The kernel function is then $K(x, y) = x_1^2y_1^2 + x_2^2y_2^2 + 2x_1x_2y_1y_2 = (x_1y_1 + x_2y_2)^2 = (x^Ty)^2$ - one can efficiently (in time proportional to the size of x, and far lesser than that of $\phi(x)$) compute the kernel function without ever computing $\phi(x)^T\phi(y)$ explicitly.

11.5.1 Mercer's theorem

Theorem 11.3 (Mercer's theorem). *Consider a function $K : \mathbb{R}^n \to \mathbb{R}^n$. For finite set $X = \{x_1, \ldots, x_n\}$, define the kernel matrix K_X by $K_X(i, j) = K(x_i, x_j)$. Then K is a valid kernel function iff K_X is positive semidefinite for all finite sets X.*

Kernel functions are not trivial to come by. One nice way to construct new kernels from existing ones is to use the fact that if K_1 and K_2 are valid kernels, then so are $K_1 + K_2$ and K_1K_2 (pointwise addition and multiplication).

11.5.2 Examples of valid kernels

1. The Polynomial Kernel: $K(x, y) = (x^T y + c)^d$ for $c \geq 0$ and $d \in \mathcal{N}$. This is a valid kernel because it is a product of valid kernels.

2. The radial basis function kernel/Gaussian Kernel

Chapter 12

The neural network

An interesting application: smelling with NNs.

12.1 The Mc-Culloch-Pitts Neuron model

This is one of the simplest model of an object that learns from data. McCulloch and Pitts proposed this model in 1943, calling it an artificial neuron (we will henceforth simply call it a neuron). Data $x = (x_1, \ldots, x_n)$ is passed to the neuron, which has weights $w = (w_1, \ldots, w_n) \in \mathbb{R}^n$, $b \in \mathbb{R}$. The neuron computes the quantity $f_{w,b}(x) = w^T x + b$, where $f_{w,b}$ called the aggregate function. The neuron then passes the result through a function g called the activation function. The output of the neuron is defined to be $g(f_{w,b}(x))$. Note that the activation of a neuron on input x is simply its output on input x.

Remark. Consider the case where g is the sign function. This neuron model is precisely the perceptron model! Indeed, in 1957, Rosenblatt proposed the perceptron model inspired by the McCulloch-Pitts neuron model. And thus, one neuron can be used to perform binary classification.

12.2 History of deep networks

- 1943: McCulloch and Pitts propose the McCulloch-Pitts neuron model.
- 1957: Rosenblatt proposes the perceptron model.
- 1960: The idea of backpropagation is proposed by a controls engineer, Kelley. It was actually developed for flight path control.
- 1969: Minsky and Papert show that the perceptron model cannot learn the XOR function.
- 1986: Rumelhart, Hinton, and Williams show that backpropagation can be used to train neural networks with hidden layers.
- 1989: The convolutional neural network is proposed by LeCun.
- 1997: The long short-term memory (LSTM) model is proposed by Hochreiter and Schmidhuber.
- 2009: Deep learning for speech recognition is proposed by Dahl.
- 2012: AlexNet wins the ImageNet competition.
- 2014: Generative adversarial networks (GANs) are proposed by Goodfellow.

- 2016: AlphaGo defeats Go world champion Lee Sedol.

The advent of deep (many hidden layers) neural networks has been made possible thanks to the following factors:

1. Vast amounts of data
2. Faster computation: specialized hardware like GPUs (and recently TPUs) for matrix computation
3. Better algorithms: better initialization, better activation functions, better optimization techniques, toolkits, libraries like TensorFlow, PyTorch, etc.

12.3 The Multi-Layer Perceptron

Okay, back to the neuron. The output of one neuron would serve quite well for regression where we need a scalar output (indeed, a single neuron with g being the identity is linear regression). Supposing we have a classification problem, say classify an input x into one of the classes $0, 1, \dots, 9$? A simple way to use a network for this is to have 10 neurons, pass the same input x to all of them, and then predict the class to be the one corresponding to the neuron with the highest output value. Indeed, this is not a bad idea, and is called (multinomial) logistic regression (see this for details).

Key Idea 1
We can use outputs from neurons to make a prediction for classification/regression problems.

In general, each neuron is expected to 'do its bit', that is, capture something interesting about the data, some pattern, etc. One can think of them as mini-perceptrons, each with its own set of weights and activations, finding a separator of their inputs. Even when they receive the same input, these neurons can perceive different aspects of the data.

To illustrate, imagine two inputs that seem similar to one neuron, as their outputs are closely related, but appear quite distinct to another neuron. This diversity arises from the different weightings applied to the input in each of these neurons. Essentially, different weights capture different separations of the data.

Supposing we could use these varied outputs in some way - give them to someone to make sense of the different separations these neurons have come up with. Here is the key idea:

Key Idea 2
The outputs of neurons can be passed as input to another neuron to possibly recognize more intricate features in the separations.

Passing the output of the d neurons as the input to another neuron forms a more complicated network of interconnected neurons - the neural network. On input x, this 'second-level' or second-layer neuron now has different details about the input x fed to it by all the 'first-layer' neurons - and these new details are, crucially, non-linear. The second-layer neuron can now separate the data on non-linear boundaries. Two second-layer neurons, with their different weights, lead to two non-linear decision boundaries.

But before we revel in glory, let's see how the final output $\hat{y}$ looks like:

$$\hat{y} = g_2(\mathbf{w}^T \mathbf{x}_1 + \mathbf{b}).$$

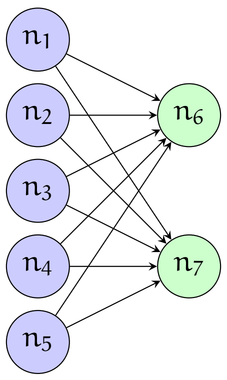


Figure 12.1: Connecting the output of neurons to the input of two other neurons.

Here $\mathbf{x}_1$ is the vector comprising the outputs of the first layer of neurons. That is, $(\mathbf{x}_1)_i = g_1(\mathbf{w}_i^T\mathbf{x} + b_i)$, where $\mathbf{w}_i$ is the weight vector of the i-th neuron in the first layer, and b_i is the bias of the i-th neuron in the first layer. If we suppose that the activations of the first layer are all identical (this typically does not hurt performance, but allows vectorization, which greatly speeds up training), we can write this in vectorized form as:

$$\mathbf{x}_1 = g_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)$$

where $\mathbf{W}_1$ is the matrix whose i-th column is $\mathbf{w}_i$, and $\mathbf{b}_1$ is the vector whose i-th entry is b_i.

Thus, we can write:

$$\begin{aligned} \mathbf{x}_0 &= \mathbf{x} \\ \mathbf{x}_1 &= g_1(\mathbf{W}_1\mathbf{x}_0 + \mathbf{b}_1) \\ \hat{\mathbf{y}} = \mathbf{x}_2 &= g_2(\mathbf{W}_2\mathbf{x}_1 + \mathbf{b}_2) \end{aligned}$$

But why stop here? Perhaps we can use the outputs of the second layer to feed into a third layer, and so on. This is the idea behind **feedforward neural networks**, also called **multi-layer perceptrons** (MLPs) (the reason for the name is fairly obvious).

Key Idea 3
We can stack multiple layers of neurons next to each other to form a *deep* neural network.

12.4 Designing a 2 hidden layered network

This is an example of a fully-connected network, i.e., all nodes in one layer is connected to every node in the next layer. The weights in the 3 layers are of the dimensions:

The weights in the 3 layers are of the dimensions:

$$\begin{aligned} \mathbf{W}^{xp} &\in \mathbb{R}^{l\times d} \\ \mathbf{W}^{pq} &\in \mathbb{R}^{m\times l} \\ \mathbf{W}^{qy} &\in \mathbb{R}^{n\times m} \end{aligned}$$

where W_{ij}^{xp} refers to the weight of the connection from the i^{th} node of the input layer to the j^{th} node of the hidden layer.

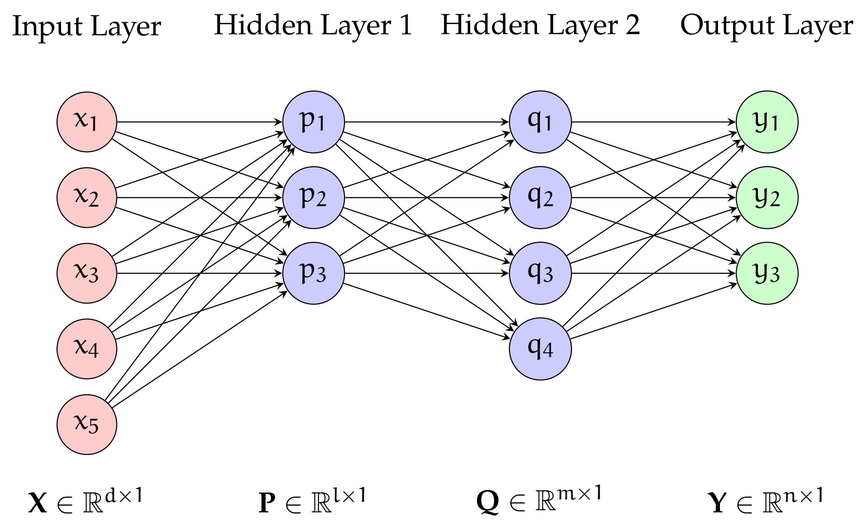

Input Layer
Hidden Layer 1
Hidden Layer 2
Output Layer
x_1
x_2
x_3
x_4
x_5
p_1
p_2
p_3
q_1
q_2
q_3
q_4
y_1
y_2
y_3
$\mathbf{X} \in \mathbb{R}^{d\times 1}$
$\mathbf{P} \in \mathbb{R}^{l\times 1}$
$\mathbf{Q} \in \mathbb{R}^{m\times 1}$
$\mathbf{Y} \in \mathbb{R}^{n\times 1}$

Figure 12.2: A (tiny) deep neural network. Note the dimensions of the input and output of each layer.

Let the activation function across all the layers be $\mathbf{g}(\cdot)$.
The relation between the layers and weights can be written as:

$$\begin{aligned}
\mathbf{P} &= g(\mathbf{W}^{xp}\mathbf{X} + \mathbf{B}^{xp}) \\
\mathbf{Q} &= g(\mathbf{W}^{pq}\mathbf{P} + \mathbf{B}^{pq}) \\
\mathbf{Y} &= g(\mathbf{W}^{qy}\mathbf{Q} + \mathbf{B}^{qy})
\end{aligned}$$

Here, $\mathbf{X}$ is our input, and $\mathbf{Y}$ is the corresponding classified output. $\mathbf{B}$ denotes the biases between each layer.

12.5 The loss function

Before we get too excited about deep neural networks, we need to figure out how to train them. That is, we need to figure out how to choose the weights and biases of the neurons so that the network does what we want it to do. To do this, as before, we need to define a loss function to tell the network how well it is doing. The loss function is a function of the weights and biases of the network, and measures how well the network is doing. The goal is to minimize the loss function.

12.6 The loss function

Before we get too excited about deep neural networks, we need to figure out how to train them. That is, we need to figure out how to choose the weights and biases of the neurons so that the network does what we want it to do. To do this, as before, we need to define a loss function to tell the network how well it is doing. The loss function is a function of the weights and biases of the network, and measures how well the network is doing. The goal is to minimize the loss function.

12.6.1 Typical loss functions

For regression problems (the output vector of the MLP is a scalar, aka there is only one output neuron), we typically use the squared error loss. Suppose the true output is y, and the output of the MLP upon input x is $\hat{y}$. Then, the squared error loss is defined by

$$\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2 \implies \mathcal{L}(\mathcal{D}) = \frac{1}{N}\sum_{i=1}^{N}(y^{(i)} - \hat{y}^{(i)})^2$$

It is convex in the weights and biases and so behaves well w.r.t gradient descent.

For classification problems, the cross-entropy loss - a generalization of the binary cross-entropy from logistic regression - is typically used. Suppose we have K classes, and so the output vector of the MLP is K-dimensional. Let y be the true label of the input x, and let $\hat{y} \in \mathbb{R}^K$ be the output of the MLP on input x. The vector $\hat{y}$ represents the probability distribution over the classes predicted for input x. The cross-entropy loss is defined as:

$$L(y, \hat{y}) = -\sum_{k=1}^{K} y_k \log \hat{y}_k \implies \mathcal{L}(\mathcal{D}) = -\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}(y^{(i)})_k \log(\hat{y}^{(i)})_k$$

How do we go from the neural network outputs to a probability distribution $\hat{y}_k = P(y = k|x)$ over the classes? We used the sigmoid for binary classification. Here we use a generalization, called the softmax function. The softmax function takes a K-dimensional vector z and outputs a K-dimensional vector $\sigma(z)$, where:

$$\sigma(z)_k = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_j}}$$

We will see how to compute the gradient of these neural networks with respect to the different weights (and thus, complete the training of neural networks) in the next lecture. We will finish this lecture with a note on the activation functions of the different neurons.

12.6.2 Common activation functions

Suppose the activation function of every neuron in the network was linear. Then the final output vector is just some linear function of the input vector, since composing linear functions gives a linear function. This is not very useful, since we want the network to be able to capture non-linearities in the data. Plus, it is then just a complicated way of doing linear regression. But hang on, see this.

Thus, we need to use non-linear activation functions. Here are some common ones:

1. **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$. A common choice for the activation function. It is smooth, and bounded. However, it suffers from the vanishing gradient problem: the gradient of the sigmoid function is very small for large values of x, and so the weights corresponding to neurons with large activations do not get updated much. This is a problem because the neurons with large activations are the ones that are most important in the network. This slows down learning significantly.

2. **Hyperbolic tangent**: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$. This is similar to the sigmoid function, but is zero-centered. This helps with learning, since the gradient is not always positive or always negative. However, it still suffers from the vanishing gradient problem.

3. **ReLU**. A very popular choice, and converges quickly.

4. **Many more!**: Leaky ReLU, ELU, SELU, etc.

Chapter 13

Backpropagation

13.1 Training the network

Now to the real deal. How do we get this to learn what the right weights are? As a function of the weights, the loss is complicated and not convex. We can still try to use gradient descent, but we have to be careful about falling into flat regions and local minima of the loss function. For simplicity, suppose we use plain old SGD:

Algorithm 2 SGD for training a neural network

```
1: Initialize weights and biases randomly
2: while not converged do
3:     Sample a minibatch of data $\mathcal{D}$
4:     Compute the gradient of the loss function $\nabla_w(L)$ on $\mathcal{D}$ with respect to the weights
5:     Update the weights and biases using the gradient. Set $w \leftarrow w - \alpha\nabla_w(L)$
6: end while
```

We now address the elephant in the room: how on earth do you compute the derivative of the loss with respect to each weight?!

13.2 Backpropagation

Set the loss function to be minimized as a loss L. We can write the dependence of the loss on the weights as a directed acyclic graph on the weights. The nodes are the weights, and the edges are the dependence of the loss on the weights. For example, if the loss is $L = L(w_1, w_2) = w_1^2 + w_2^2$, then the graph is:

$$\begin{tikzcd}[column sep=small]
 & & & L = a + b & & & \\
 & a = w_1^2 \arrow[urr] & & & & b = w_2^2 \arrow[ull] & \\
w_1 \arrow[ur] & & w_1 \arrow[ul] & & w_2 \arrow[ur] & & w_2 \arrow[ul]
\end{tikzcd}$$

Suppose that the loss depended on a weight w by $\mathcal{L} = g(u)$, where $u = f(x)$. We have, by the chain rule, the expression

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial u}\frac{\partial u}{\partial w}$$

In our case above, $g(a) = a + b$, $a = w_1^2$. We get

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial a}\frac{\partial a}{\partial w_1} = 1 \cdot 2w_1$$

. Similarly, $\frac{\partial L}{\partial w_2} = 2w_2$. And we're done. In a more complicated example, how will this work? We need the multuvariable chain rule.

Theorem 13.1 (Multivariable chain rule). *Suppose* $f = f(y_1, \dots, y_n)$ *is a function. For some* i, *suppose* $y_i = g(x_i)$. *Then*

$$\frac{\partial f}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial f}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

In our case, $f = L$ and the y_is are the immediate children of the loss in the DAG. x_i is a weight (which is at the leaf of the DAG).

The theorem above tells us that we can compute $\frac{\partial L}{\partial w}$ if we know $\frac{\partial u}{\partial w}$ for every child u of L. The relations $\frac{\partial L}{\partial u}$ are local, and are simply functions of the children of L. (For example, if $L(u_1, \dots, u_k) = u_1^2 + u_2 * \tanh u_3$, then $\frac{\partial L}{\partial u_1} = 2u_1$, $\frac{\partial L}{\partial u_2} = \tanh u_3$, $\frac{\partial L}{\partial u_3} = u_2/\cosh^2 u_3$.)

This is a recursive procedure, and thus we can compute the derivatives by starting at the leaves and working our way up to the root. This, in essence, is backpropagation.

13.3 The algorithm

The algorithm involves two passes: a forward pass and a backward pass. The forward pass computes the values of the nodes in the DAG. The backward pass computes the derivatives of the loss with respect to the weights (and thus we have the gradient).

13.3.1 Forward pass

The forward pass is simple. We simply compute the values of the nodes in the DAG in their topological order.

Algorithm 3 forward pass

```
1: The values of the leaves are the input values and the values of the weights.
2: for each node $u$ in the DAG in topological order do
3:     compute $u.val = f(u.child_1.val, u.child_2.val, \dots, u.child_k.val)$
4: end for
```

13.3.2 Backward pass

Here we use the computed values to compute the gradient of every node with respect to the leaves that vary (i.e. the gradient of a node wrt the input is 0, and we do not bother about these).

While building the DAG itself, we store the dependences of the parent $di = \frac{\partial u}{\partial(u.child_i)}$ on its children as **functions**. These depend simply on the children's values. Note that we use some of the input's values as required (for example, in $w^T x$) in the functions di but we do not include them as children because they are not variables that we are differentiating with respect to.

Then the algorithm proceeds recursively like so:

Algorithm 4 backward pass

```
1: The function $dw$ for the leaves (weights) is set to $1$.
2: for each node $u$ in the DAG in topological order do
3:     compute $u.grad = \sum_{i=1}^{k} u.di \times child_i.grad$ (where $\star.grad$ is a vector with dimension equal to the number of weights)
4: end for
```

13.4 Vectorized implementation of backpropagation

13.5 Regularization

13.5.1 L2 regularization

Introduce another loss term that penalizes the squared magnitude of **all** parameters. That is, for every weight w, add the term λw^2 to the loss. This is called L2 regularization. The intuition, as before, is that this will penalize large weights, and thus prevent overfitting.

13.5.2 Dropout

During training, randomly set the activations of some neurons to 0. This is called dropout. This prevents the network from relying too much on any one neuron, and thus prevents overfitting. Theoretical guarantees make use of the fact that the neural network is an average of exponentially many smaller networks.

We do not dropout any of the input or output neurons, of course. An important point: at test time, we do not use dropout. Instead, we multiply the activations of each neuron by the probability that it was not dropped out during training - the expected value of the activation is used.

Remark. An extreme version of dropout is to randomly drop out entire layers of neurons. This is called dropconnect. This makes use of residual connections, which connect the input of a layer to the output of a layer that is not the immediate previous layer.

13.5.3 Early stopping

Stop training when the validation loss starts increasing. This is called early stopping. This is a form of regularization because it prevents the network from overfitting to the training data (we stop training before it can do so).

Chapter 14

Learning Rate Scheduling

Is vanilla SGD the best way to train a neural network? What is a *good* learning rate? Figure 14.1 shows the effect of the learning rate on the loss function.

14.1 Momentum

There are two main motivations for momentum, which are limitations of SGD:

- SGD can have large weight updates on complex loss landscapes. The gradient is all over the place and the weight updates make training unstable.
- At flat points like saddle points, SGD can get stuck. The gradient is zero, and the weight updates are zero. How do we get out of a flat point?

SGD with momentum addresses these issues. Consider $0 \leq \beta < 1$ and define a velocity term v_t as follows:

$$v_t = \beta v_{t-1} + \eta(\nabla_w L)(w_t)$$

That is, the velocity is an exponentially decaying average of the gradients:

$$\begin{aligned} v_t &= \eta(\nabla_w L)(w_t) + \beta v_{t-1} \\ &= \eta(\nabla_w L)(w_t) + \beta\eta(\nabla_w L)(w_{t-1}) + \beta^2 v_{t-2} \\ &= \eta \sum_{i=0}^{t} \beta^i (\nabla_w L)(w_{t-i}) \end{aligned}$$

The weight update is then:

$$w_{t+1} = w_t - v_t$$

The parameter β is a measure of how much we value old gradients. A value of 0 results in SGD, and a value close to 1 results in a very smooth (but not very relevant to the current step) gradient.

Notice how this allows us to escape flat points. The only issue, is that it tries to escape minima as well! Since gradients are small near a minima, the velocity gradually decreases, but it is true that one will overshoot first and then come back. The velocity will likely be small when it comes back, and so it will not overshoot again.

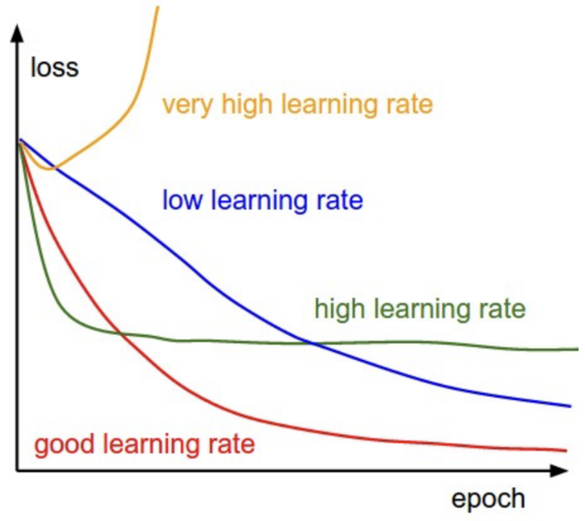


Figure 14.1: What is a good learning rate?

14.2 ADAgrad

The idea behind ADAgrad (adaptive gradient) is to have a different learning rate for each parameter - learning rates adapt to different parameters. The motivation is that some parameters are more sensitive than others, and so we should update them more slowly. Define the operator $\odot$ to be elementwise multiplication. We maintain the vector s_t defined by

$$s_t = s_{t-1} + g_t \odot g_t$$

where $g_t = \nabla_w L(w_t)$ is the gradient at time t. The weight update is then:

$$w_{t+1} = w_t - \eta \frac{1}{\sqrt{s_t} + \epsilon} \odot g_t$$

where ϵ is a small constant to avoid division by zero.

Putting it in one line, the learning rate for the ith parameter is:

$$\eta_i = \frac{\eta}{\sqrt{\sum_{t=1}^{T} (\nabla_w L(w_t))_i^2} + \epsilon}$$

In some sense this is a normalization of the gradient. The weight update is then:

$$(w_{t+1})_i = (w_t)_i - \eta_i((\nabla_w L)(w_t))_i$$

Remark. This is an algorithm that is backed by theory. See Duchi et al. (2011) for more details.

There is an obvious problem with this algorithm. The denominator is a sum of squares, and so it keeps increasing. This means that the learning rate keeps decreasing, and so the algorithm will eventually stop learning. This is solved by the next algorithm.

14.3 RMSProp

We continue to maintain vector s_t, except

$$s_t = \gamma s_{t-1} + (1 - \gamma) g_t \odot g_t$$

The update step is identical. The decay allows us to forget old gradients, and so the learning rate does not necessarily keep decreasing.

Remark. This was not a paper! This was actually proposed in a Coursera course by Geoff Hinton.

14.4 ADAM

The best of both worlds. We maintain two vectors m_t and s_t defined by

$$\begin{aligned} v_t &= \beta v_{t-1} + (1-\beta) g_t \\ s_t &= \gamma s_{t-1} + (1-\gamma) g_t \odot g_t \end{aligned}$$

where $g_t = \nabla_w L(w_t)$. The update step is then:

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{s_t + \epsilon}} \odot v_t$$

The actual paper does something slightly different: the weight update is

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{s}_t + \epsilon}} \odot \hat{v}_t$$

where

$$\begin{aligned} \hat{v}_t &= \frac{v_t}{1-\beta^t} \\ \hat{s}_t &= \frac{s_t}{1-\gamma^t} \end{aligned}$$

This is to correct for the fact that the vectors v_t and s_t are initialized to zero, and so they are biased towards zero. Okay, so what?

We would like to keep v_t always a moving average of gradients. We have:

$$v_t = (1-\beta)\left[\sum_{i=0}^{t-1} \beta^i g_{t-i}\right]$$

with sum of coefficients being $(1-\beta^t)$. For t large, this is not a problem. But it does help to divide by $(1-\beta^t)$ to make the velocity a moving average of gradients.

Chapter 15

Convolutional Neural Networks

Consider the problem of recognizing an object in an image. Humans are able to do this very well, but how do we get a computer to do this? We would like the following properties from a model that learns to do this:

- **Translation invariance**. If we move the object around in the image, we would like the model to still recognize it.
- **Locality**. ?
- **Parameter-efficiency**. We would like to learn as few parameters as possible.

A standard feedforward network where the input is the flattened image ticks none of these boxes. Neural networks with a twist - convolutions are required. From this point on, we shall continue to modify the vanilla neural network for various tasks.

15.1 Cross-correlation

Definition 15.1. Let x and w be vectors of length n. The **cross-correlation** of x and w is defined as:

$$(x \star w)_i = \sum_{j=1}^{n} x_{i+j-1} w_j$$

Chapter 16

Recurrent Neural Networks

There is a paper that finds Automata that are equivalent to RNNs, authors: Weiss et al. 2018. Name of the paper: Extracting Automata from Recurrent Neural Networks Using Queries and Counterexamples.

Chapter 17

Unsupervised Learning: Clustering

Consider a set of points $X = \{x_1, \ldots, x_n\}$ in $\mathbb{R}^d$. We want to find similar (defined by close-by according to some metric on $\mathbb{R}^d$) points and group them together (assign them the same class). This is called *clustering* (into classes).

Definition 17.1 (Clustering). Let $X = \{x_1, \ldots, x_n\}$ be a set of points in $\mathbb{R}^d$. A *clustering* of X is a partition of X into k subsets $C_1, \ldots, C_K$ such that - in some well-defined sense - points in the same subset are similar and points in different subsets are dissimilar.

Of what utility is clustering? Consider the following applications:

1. *Market segmentation*: Given a set of customers, group them into clusters based on their purchasing behavior.
2. *Social network analysis*: Given a set of users, group them into clusters based on their social interactions.
3. *Image segmentation*: Given an image, group its pixels into clusters based on their color.
4. *Astronomical data analysis*: Given a set of stars, group them into clusters based on their brightness.

17.1 k-means Clustering

A simple algorithm for clustering into k clusters. The intuition is the following: every datapoint is close to the centre of the mean of its assigned cluster. This means that once we fix the *cluster centres*, it is very easy to complete the cluster (assign each point to the cluster whose centre is closest to it).

Here is the key point: a good heuristic for the cluster centres is to take the mean of the points in the cluster. Indeed, the mean minimizes the sum of squared distances to all the points in the cluster (does that hint at what a good loss could be?).

So what we could do is: pickup random points as the cluster centres, and repeat the following two steps until convergence:

1. Assign each point to the cluster whose centre is closest to it.
2. Update each cluster centre to be the mean of the points in the cluster.

This alternates between two steps: *assignment* (computing the cluster corresponding to a set of clusters) and *update* (update the cluster centres based on the cluster). This is called the *K-means algorithm*. In a way, it is similar to Generalized Policy Iteration (GPI) in Reinforcement Learning - where we alternate between *policy evaluation* (computing the values corresponding to a policy) and *policy improvement* (update the policy based on the values). Indeed, all these algorithms are instances of *Expectation Maximization* (EM).

Let us make this precise. Let $X = \{x_1, \dots, x_n\}$ be a set of points in $\mathbb{R}^d$, a clustering $C_1, \dots, C_k$ and a set of cluster centres $\mu_1, \dots, \mu_k$. Define $C_j^i := \mathbb{1}_{[x_i \in C_j]}$.

Goal: find $\mu_1, \dots, \mu_k$ such that the sum of squared distances of each point to its cluster centre is minimized. That is, we want to solve the following optimization problem:

$$\min_{\mu_1,\dots,\mu_k} \min_{C_1,\dots,C_k} \sum_{i\in[n]} \sum_{j\in[k]} C_j^i \|x_i - \mu_j\|^2 \tag{17.1}$$

yielding the loss:

$$\mathcal{L} = \sum_{i\in[n]} \sum_{j\in[k]} C_j^i \|x_i - \mu_j\|^2 = \sum_{j\in[k]} \sum_{x\in C_j} \|x - \mu_k\|^2$$

Suppose that we fix the clustering $C_1, \dots, C_k$. Then the optimal $\mu_1, \dots, \mu_k$ are given by setting the derivative $\frac{\partial \mathcal{L}}{\partial \mu_j} = 0$ for all $j \in [k]$. We have

$$0 = \frac{\partial \mathcal{L}}{\partial \mu_j} = 2 \sum_{x\in C_j} (x - \mu_k) \implies \mu_j = \frac{1}{|C_j|} \sum_{x\in C_j} x$$

that is, given a cluster, the optimal cluster centre is the mean of the points in the cluster - precisely the update step in the K-means algorithm.

Now suppose that we fix the cluster centres $\mu_1, \dots, \mu_k$. Then the optimal $C_1, \dots, C_k$ are given by assigning each point to the cluster whose centre is closest to it (We minimize the loss for fixed μ_i over each z_i - the label of each x_i).

Clearly, the minimum is achieved at

$$z_i = \arg\min_{j\in[k]} \|x_i - \mu_j\|^2$$

(so $C_j^i = \mathbb{1}_{\left[j = \arg\min_{j\in[k]} \|x_i - \mu_j\|^2\right]}$ for each i, j) Thus, given the cluster centres, the optimal clustering is the one that assigns each point to the closest cluster centre - precisely the assignment step in the k-means algorithm.

17.2 An algorithm for k-means

Algorithm 5 k-means

```
1: Initialize $\mu_1, \dots, \mu_k$ randomly
2: while not converged do
3:     Assign each point to the cluster whose centre is closest to it
4:     Update each cluster centre to be the mean of the points in the cluster
5: end while
```

Another algorithm that makes explicit the two alternating steps is (the set of possible cluster centres is Θ, and of the possible assignments (clusters) is Φ):

Algorithm 6 k-means

```
Initialize $\theta \in \Theta$ randomly
flag = True
prev = -1
while True do
    if flag then
        $\phi = \arg\min_{\phi' \in \Phi} \mathcal{L}(\theta, \phi')$ // Assign each point to the cluster closest to it
    else
        $\theta = \arg\min_{\theta' \in \Theta} \mathcal{L}(\theta', \phi)$ // Update each cluster centre to be the mean of the cluster
    end if
    flag = not flag
    new = $\mathcal{L}(\theta, \phi)$
    if new == prev then
        break
    end if
end while
```

17.3 Convergence of k-means

Convergence:

Convergence in finitely many steps: there are only finitely many clusterings and cluster centres, so the algorithm must converge in finitely many steps.

k-means is doing gradient descent on the loss function $\mathcal{L}$!

(However, it is not guaranteed to converge to the global minimum. In fact, it is NP-hard (need to check a lot of the k^n possible clusterings) to find the global minimum of the loss.)

Chapter 18

Gaussian Mixtures: the generative model for k-means

Let us take a different route - the route of probabilistic modelling to solve the clustering problem. We will see that the k-means algorithm is doing MLE on a Gaussian Mixture Model (GMM).

Definition 18.1 (Generative model). A *generative model* is a model that tries to *learn* the distribution of the data. Typically, this is done by estimating either the joint $p(x, y)$ or the two distributions $p(x|y)$ and $p(y)$ or $p(x)$ and the conditional $p(y|x)$.

Definition 18.2 (Discriminative Model). A *discriminative model* is a model that tries to *learn* the conditional distribution $p(y|x)$ - it tries to learn to discriminate between the outputs y given an x.

18.1 Motivation

The probabilistic model for regression was a linear model with additive Gaussian noise - that is, the underlying generative model "looked linear" in a certain sense. What does the underlying generative model for clustering look like?

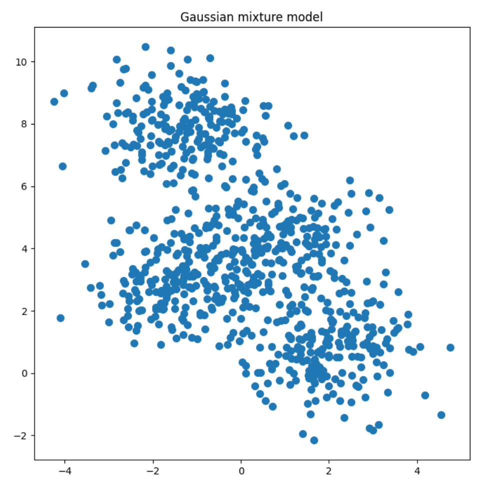
One could think about how data for clustering may have been generated by a friendly alien: The alien decides that there will be k clusters $\{1, \dots, k\}$, and picks k points $\mu_1, \dots, \mu_k$ in $\mathbb{R}^d$ as the cluster centres. He also picks a probability distribution π over the k clusters.

Then, when the alien wants to generate a datapoint, he simply picks a cluster j with probability π_j and then picks a point x from the distribution $\mathcal{N}(\mu_j, I)$. (For simplicity of exposition, we assume that the covariance of the Gaussian is the identity - it's not a bad assumption).

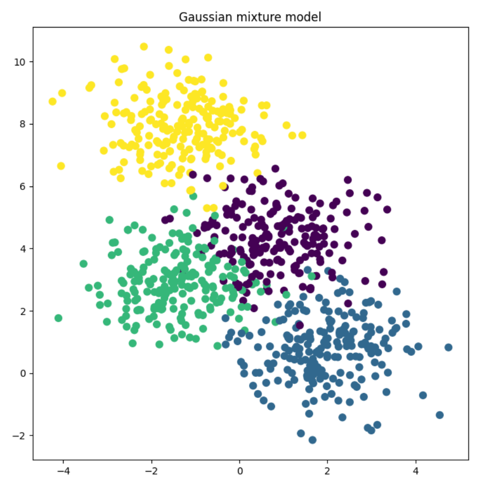
Think about it - it is a very reasonable modelling for a clustering generative model. The values π_j indicate the density of each cluster and the values μ_j indicate the centre of each cluster. The model is essentially saying that the clusters are isotropic Gaussian blobs about the μ_j with density π_j.

18.2 Formal definition

Suppose that the dataset $\mathcal{D} = \{x_1, \dots, x_n\}$ was sampled from the parameterized generative model defined below with some parameters. We use MLE to estimate the parameters of the model (which, in the limit of infinite training data, will be exact - the ML estimate is consistent).



(a) Alien generates $n = 800$ datapoints with $k = 4$ clusters.



(b) The clusters that the alien used for each datapoint colored for clarity

Figure 18.1: Alien generates data from a GMM

- Pick a cluster $j \in [k]$ with probability π_j. We write the random variable corresponding to the cluster chosen as z.
- Pick x from the distribution $\mathcal{N}(\mu_j, I)$ (notice how this depends on j). For simplicity of exposition, we assume that the covariance of the Gaussian is the identity.

Essentially, we have parameterized the generative model by μ_j and π_j that define the joint by:

$$p(z = j) = \pi_j$$
$$p(x|z = j) = \mathcal{N}(\mu_j, I)$$

A model with the above parameterization is called a Gaussian Mixture Model (GMM). The joint distribution is given by:

$$p(x, z) = p(z)p(x|z) = \pi_j \mathcal{N}(\mu_j, I)$$

Okay. We'd like to estimate parameters of our model and find the one that matches the underlying GMM of the data. In the unsupervised learning situation, we are not given the labels z (which cluster each point belongs to): we need to estimate the parameters μ_j and π_j simply from the data $\mathcal{D} = \{x_1, \dots, x_n\}$.

To this end, we marginalize out z to work with the probabilities

$$p(x) = \sum_{j \in [k]} p(x, z = j) = \sum_{j \in [k]} \pi_j \mathcal{N}(\mu_j, I)$$

The log-likelihood of the data is given by:

$$\mathcal{L}(\mu, \pi; \mathcal{D}) = \sum_{i \in [n]} \log p(x_i) = \sum_{i \in [n]} \log \sum_{j \in [k]} \pi_j \mathcal{N}(\mu_j, I)$$

A very messy expression. $p(x)$ is not a great term to put in a log. Let's take a small detour to think about how we can simplify the expression.

Suppose we were given labels $\mathcal{Z} = \{z_1, \ldots, z_n\}$ (generated by the underlying generative model) corresponding to each point x_i. Then the log-likelihood would be given by:

$$\begin{aligned}\mathcal{L}(\mu, \pi; \mathcal{D}, \mathcal{Z}) &= \sum_{i\in[n]} \log p(x_i, z_i | \mu, \pi) = \sum_{i\in[n]} \log \pi_{z_i} \mathcal{N}(\mu_{z_i}, I) \\ &= \sum_{i\in[n]} \log \pi_{z_i} + \sum_{i\in[n]} - \log \sqrt{2\pi} + \sum_{i\in[n]} -\frac{1}{2} \|x_i - \mu_{z_i}\|^2\end{aligned}$$

Let's write the likelihood in a more symmetric way:

$$\mathcal{L}(\mu, \pi; \mathcal{D}, \mathcal{Z}) = \sum_{j\in[k]} \sum_{i\in[n]} \mathbb{1}_{[z_i=j]} \log \pi_j + \sum_{i\in[n]} - \log \sqrt{2\pi} - \frac{1}{2} \sum_{j\in[k]} \sum_{i\in[n]} \mathbb{1}_{[z_i=j]} \|x_i - \mu_j\|^2$$

Maximizing the likelihood yields (for pik, cant differentiate, so use lagrange multipliers):

$$\mu_j = \frac{\sum_{i\in[n]} \mathbb{1}_{[z_i=j]} x_i}{\sum_{i\in[n]} \mathbb{1}_{[z_i=j]}} \tag{18.1}$$

$$\pi_j = \frac{\sum_{i\in[n]} \mathbb{1}_{[z_i=j]}}{n} \tag{18.2}$$

Notice how the ML estimate for the cluster centre is the mean of the points in the cluster - the update step in k-means. Since the labels are given to us, the clusters are fixed, so the k-means reduces to just the one step of updating the cluster centres. Thus, the k-means algorithm is just an ML estimate for a GMM model on the underlying data.

This is not a coincidence - just like in linear regression, the log of the Gaussian gives the squared distance which is precisely the loss we are trying to minimize in the k-means algorithm.

Back to reality. Of course, we are not given the labels. But we can still use the above expression to simplify the log-likelihood; interpreting the result as a sort of conditional over the values in $\mathcal{Z}$. Introduce the *responsibilities* or *occupancies* γ_j^i of each cluster j for each point i (the probability that point i belongs to cluster j):

$$\gamma_j^i = p(z = j | x_i) = \frac{p(x_i | z = j) p(z = j)}{p(x_i)} = \frac{\pi_j \mathcal{N}(\mu_j, I)}{\sum_{j'\in[k]} \pi_{j'} \mathcal{N}(\mu_{j'}, I)}$$

Remark. In case the covariance is not the identity - suppose it is a diagonal matrix for each cluster - that simply brings more parameters for the MLE, the rest works out similarly.

-> maximize wrt uk, pik assuming that he gamma variables are independent of these -> and then update the gamma variables

18.3 Expectation Maximization for GMMs

A general algorithm applying the same idea as above is called the Expectation Maximization (EM) algorithm. It is an iterative algorithm that alternates between two steps: the *expectation* step and the *maximization* step.

The algorithm finds a *lower bound* on the true log-likelihood of the data, and tries to maximize this lower bound. The lower bound is called the *evidence lower bound* (ELBO) and is given by:

$$\mathbb{E}_{z|x,\theta^{(t)}} [\log p(x, z|\theta)] - \mathbb{E}_{z|x,\theta^{(t)}} [\log q(z)]$$

Algorithm 7 Expectation Maximization

```
1: Initialize parameters $\theta$
2: while not converged do
3:     E-step: Compute the expected value of the log-likelihood wrt the conditional distribution
       of the latent variables given the data and the current estimate of the parameters:
           $\mathbb{E}_{z|x,\theta^{(t)}}[\log p(x, z|\theta)]$
4:     M-step: Update latent variables by maximizing the expected log-likelihood wrt parameters:
           $\theta^{(t+1)} = \arg\max_{\theta} \mathbb{E}_{z|x,\theta^{(t)}}[\log p(x, z|\theta)]$
5: end while
```

18.3.1 The EM algorithm to fit GMMs to data

1. Initialize $\mu_k \leftarrow \mu_k^{\text{init}}$, $\pi_k \leftarrow \pi_k^{\text{init}}$ for $k \in [K]$.
2. **E-step**: Compute the responsibilities $\gamma_{k,i}$ for each point i and cluster j:
$$\gamma_{k,i} = p(z = k|x_i) = \frac{\pi_k \mathcal{N}(\mu_k, I)}{\sum_{k' \in [k]} \pi_{k'} \mathcal{N}(\mu_{k'}, I)}$$
3. **M-step**: Estimate μ_k, π_k using the maximum likelihood estimates that we found on keeping the $\gamma_{k,i}$ fixed:
$$\mu_k = \frac{\sum_{i \in [n]} \gamma_{k,i} x_i}{\sum_{i \in [n]} \gamma_{k,i}}$$
$$\pi_k = \frac{\sum_{i \in [n]} \gamma_{k,i}}{n}$$
4. Repeat steps 2 and 3 until convergence.

It is guaranteed to converge to a local optimum of the log-likelihood. The proof is a bit involved, but the intuition is that the log-likelihood is a concave function of the parameters, and the EM algorithm is a coordinate ascent algorithm on the log-likelihood.

18.4 The EM algorithm for k-means

Chapter 19

Dimensionality reduction

Most datapoints today have a very large number of features - that is, the vectors x are in a high-dimensional space. Images are an obvious example - a 100×100 color image has $30,000$ features. This is an obvious issue for many algorithms - iterative optimization algorithms like k-means are quite slow in high dimensions; in high dimensions, the data is very sparse and the curse of dimensionality makes it hard to find meaningful clusters.

Even for exact regression, the matrices that need to be multiplied become rather large, and since matrix multiplication is lower bounded by $\Omega(n^2)$, this becomes a problem.

In this chapter, we will look at two methods of dimensionality reduction: *Principal Component Analysis* (PCA) and *t-SNE*.

19.1 Principal Component Analysis

The most popular dimensionality reduction technique. It is a linear technique - that is, it tries to find a linear subspace of the original space that *captures most of the variance* in the data. It is also a *projection* technique - that is, it projects data onto this subspace.

19.1.1 Motivation

It is typically true that though the input space may have very large dimensionality, the data may be concentrated in a lower-dimensional subspace.

Thus, we aim to compute this subspace. We will do this by finding a set of orthonormal vectors $u_1, \dots, u_k \in \mathbb{R}^d$ such that the projection of the data onto the subspace spanned by these vectors is as close in spirit as possible to the original data.

That is, we transform x to

$$z = U^T x \in \mathbb{R}^k$$

where $U = [u_1, \dots, u_k] \in d \times k$ with $k \ll d$. Note that $U^T U$ is the Grammian of the vectors $u_1, \dots, u_k$ and is the identity matrix since the vectors are orthonormal. Thus U is one-sided orthogonal.

The transformation is a projection:

$$z = \begin{bmatrix} u_1^T \\ \vdots \\ u_k^T \end{bmatrix} x = \begin{bmatrix} u_1^T x \\ \vdots \\ u_k^T x \end{bmatrix}$$

.

19.1.2 What is the subspace, exactly?

19.1.3 Objective

We wish to minimize the *reconstruction error*

$$\|x - Uz\|^2 = \left\| x - \sum_{i=1}^{k} u_i z_i \right\|^2$$

where $z_i = u_i^T x$.

A second objective of dimensionality reduction could be to maximize the variance of the projeected data:

$$\max_{U^T U = I_k} \sum_{i=1}^{k} \operatorname{var}(z_i)$$

For PCA, both of these can be shown to be equivalent!

We have

$$\begin{aligned} x &= UU^T x + (I - UU^T)x \\ \implies \|x\|^2 &= \left\|UU^T x\right\|^2 + \left\|x - UU^T x\right\|^2 \\ \implies \mathbb{E}\left[\|x\|^2\right] &= \mathbb{E}\left[\left\|UU^T x\right\|^2\right] + \mathbb{E}\left[\left\|x - UU^T x\right\|^2\right] \end{aligned}$$

thus the variance of the projected data is maximized iff the reconstruction error is minimized.

19.1.4 Finding the subspace

We use objective 2. We want to maximize

$$\sum_{i=1}^{n} \left\|x_i^T U\right\|^2 = U^T \left(\sum_{i=1}^{n} x_i x_i^T\right) U = U^T S U.$$

It is standard to see that the maximum is achieved when U is the matrix of eigenvectors of S corresponding to the k largest eigenvalues. That's PCA.

Remark. The matrix S is called the *covariance matrix* of the data. It is a symmetric matrix, and its diagonal entries are the variances of the features. The off-diagonal entries are the covariances of the features.

Remark. Diagonalization to find eigenvectors can be very expensive (at least quadratic in the dimensionality). A faster algorithm uses the singular value decomposition: the top few eigenvectors of the covariance matrix are the same as the top few singular vectors of the data matrix.

Chapter 20

Ensemble methods

20.1 Aside: A few interesting papers

- How to avoid machine learning pitfalls: a guide for academic researchers
- Pen and Paper exercises in Machine Learning
-

20.2 A brief look at Autoencoders

Autoencoders are a class of neural networks that are trained to reconstruct their input. They are used for dimensionality reduction, denoising, and generative modeling. To avoid trivial solutions, we usually impose a bottleneck in the network, forcing it to learn a compressed representation of the input.

A linear autoencoder in the most general case is a pair of matrices $W_1 \in \mathbb{R}^{d \times k}$ and $W_2 \in \mathbb{R}^{k \times d}$. If $k \ll d$, we achieve dimensionality reduction. The function $f : \mathbb{R}^d \to \mathbb{R}^k$ defined by $f(x) = W_1 x$ is called the encoder, and the function $g : \mathbb{R}^k \to \mathbb{R}^d$ defined by $g(x) = W_2 x$ is called the decoder. The autoencoder is the composition $g \circ f$. The goal is to find W_1 and W_2 such that the autoencoder is a good approximation of the identity function, even with the bottleneck.

The loss is usually the mean squared error between the input and the output (called the reconstruction error):

$$\mathcal{L}(x, \tilde{x}) = \|x - \tilde{x}\|^2$$

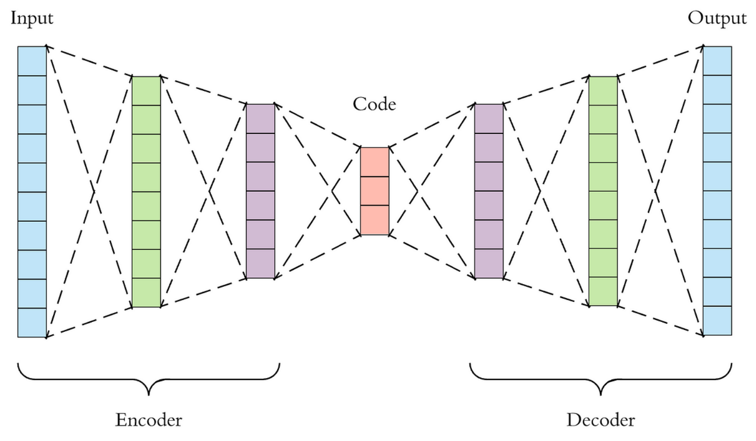


Figure 20.1: A linear autoencoder. Taken from here.

It can be shown that the minimum reconstruction error for $k < d$ is achieved by $W_1 = W_2^T = \Sigma_k^{-1/2} U_k^T$, where U_k is the matrix of the first k eigenvectors of the covariance matrix XX^T of the data. k is a hyperparameter that can be chosen by validation on the validation set.

Nonlinear autoencoders are, of course, just when the functions f and g are nonlinear. The most common choice is to use neural networks with nonlinear activation functions. Another way is to use nonlinear kernels and then use PCA - this is called kernel PCA.

20.3 Boosting

A very popular ensemble learning algorithm. The intuition is as filows: Can we start with a weaker model (a learner that is slightly better than the random model in expectation) and make it stronger by focusing on the difficult/hard to classify examples?

The main template of boosting algorithms is:

1. Train a weak learner on the training data.
2. Computer the error of the weak learner on the training data.
3. Increase the weights of the examples that were misclassified by the weak learner. This is the *boosting* step.
4. Go back to step 1.

This is a greedy algorithm over the space of possible functions. The weights of the examples are initialized to $1/N$ and are increased if the example is misclassified. The loss function is a weighted sum of the loss of each example.

Remark. Boosting algorithms are traditionally called meta-algorithms because they take as *argument* any algorithm for classifation. The boosting algorithm itself is not a classifier, but rather a procedure that takes a classifier and returns a better one.

20.3.1 AdaBoost

The most popular boosting algorithm. The algorithm is described in Algorithm 8 for binary classification. Here is the remarkable thing: the training error of AdaBoost is guaranteed to decrease exponentially fast with the number of iterations T as long as ϵ_t's correspond to learners better than chance. Even more remarkably, there are guarantees on its generalization error as well.

Algorithm 8 AdaBoost for Binary Classification

```
1:  Initialize $w_i = 1/N$ for $i = 1, \ldots, N$
2:  Initialize weak learner $h_t$ to be a decision stump: $h_t(x) = \begin{cases} 1 & \text{if } x_j \leq \theta \\ -1 & \text{if } x_j > \theta \end{cases}$
3:  for $t = 1, \ldots, T$ do
4:      Train a weak learner $h_t$ on the training data with weights $w_i$
5:      Compute the error $\epsilon_t$ of $h_t$ on the training data:
            $$\epsilon_t = \sum_{i=1}^{N} w_i \mathbb{1}_{[y_i \neq h_t(x_i)]}$$
6:      Compute $\alpha_t = \frac{1}{2} \log\left(\frac{1-\epsilon_t}{\epsilon_t}\right)$. Note that $\alpha_t > 0$ iff $\epsilon_t < 1/2$.
7:      Update the weights: $w_i \leftarrow w_i \exp\left(\alpha_t (-1)^{\mathbb{1}_{[y_i \neq h_t(x_i)]}}\right)$
8:      Normalize the weights: $w_i \leftarrow \frac{w_i}{\sum_{j=1}^{N} w_j}$
9:  end for
10: Output the boosted classifier $H(x) = \sigma\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right)$
```

Chapter 21

Search

We move now to artificial intelligence, more generally. So far, we have seen reflex-based models, where given an input, we make a prediction using a set of rules (the trained model). All computations were *forward*, with no *backtracking*. We now look at state-based models: state information (the representation of the environment and current calculation), action information, goals (the 'input' to the model), and a *plan*. In some sense, state-based models are a bit like rational models.

21.1 The search problem

For now, we operate under the assumption that the environment is fully known. A search problem can be defined using:

1. A set of *states* S - called the state space, and a set of actions A (the action space).
2. A start state $s \in S$, and a goal state $f \in S$.
3. A transition funtion that maps $s \xrightarrow{a} s'$ on action a.

A *solution* is a sequence of states from the start state to the goal state. An optimal solution is one with minimum cost.

21.1.1 The frontier

At any time, there are three sets of states: the explored set E, the frontier F, and the unexplored set U. The explored set is the set of states that have been visited and expanded from, the frontier/fringe is the set of states that have been visited but not expanded, and the unexplored set is the set of states that have not been visited. The frontier is the set of states that are on the boundary between the explored and unexplored sets.

A step in the search would look like so: pick a state from the frontier based on some *strategy* (based on some sort of priority, or randomly), expand it to child nodes. Move the chosen frontier state to explored, and add the children to the frontier.

21.2 Uninformed search

This kind of search algorithms do not use any information about the goal state. They do not use some sort of proximity to the goal to guide their strategy. DFS - expands the 'deepest' node first/node on top of the stack - and BFS - expand shallowest node first/node on top of the queue - on the state-transition graph are examples of uninformed search algorithms.

Another example is uniform cost search, which expands the node with the lowest path cost (cost of the path from the start state to this node). The strategy is to pick up the lowest cost node from the frontier (using a priority queue, say). Why the name? - because we explore nodes in the order of their cost: all nodes with the same cost are visited in turn, and then we move on to the next cost.

Algorithm 9 Uniform cost search

```
 1: $\mathtt{explored} \leftarrow \varnothing$
 2: $\mathtt{frontier} \leftarrow \{s\}$
 3: $\mathtt{unexplored} \leftarrow S \setminus \{s\}$
 4: while $\mathtt{frontier} \neq \varnothing$ do
 5:     $n, p \leftarrow \mathtt{pop}(\mathtt{frontier})$ // pop the lowest cost node. $p$ stores the cost of a minimal path to $n$
 6:     $\mathtt{explored} \leftarrow \mathtt{explored} \cup \{n\}$
 7:     if $n = f$ then
 8:         return $n$
 9:     end if
10:     for $a \in A$ do
11:         $n' \leftarrow$ child of $n$ on action $a$
12:         if $n' \notin \mathtt{explored}$ then
13:             $\mathtt{frontier.update\_priority}(n', p + \mathtt{cost}(n, a, n'))$ // the $+$ operator need to satisfy
    the triangle inequality for the proof of optimality
14:         end if
15:     end for
16: end while
17: return $\mathtt{failure}$ // no solution found
```

Note that all costs must be non-negative and the $+$ operator must satisfy the triangle inequality for the proof of optimality to work.

Uninformed search is optimal, yes, but there is an important problem here: UCS may readliy move "away" from the goal state, because they do not bother with it (and are not informed about their distance from the goal anyway). Informed search essentially provides a fix to this problem.

21.3 Best-first search

The ordering of nodes in the frontier is crucial. Suppose the order is based on a function $f : S \to \mathbb{R}$. The node n in teh frontier with the least value of $f(n)$ is chosen to be expanded. In general, this is best-first search (the node in the frontier with the best value of f is chosen to be expanded). The function f is called the evaluation function.

UCS is a special case of best-first search, where $f(n) = p(n)$, the min path cost from the start state to n. A best-first search that avoids the problem with UCS is greedy best-first search, where $f(n) = h(n)$, the *estimated* cost from n to the goal state. It is quite good if the estimate is good. But note that it is not optimal, because it does not take into account the cost of the path from the start state to n.

21.3.1 A* search

This algorithm picks up the best of both worlds: it is fast, and it is optimal if the heuristic h (estimate for the distance to the goal) is consistent. The evaluation function used is $f(n) = g(n) + h(n)$. A consistent heuristic satisfies a certain triangle inequality: $h(n) \leq c(n, a, n') + h(n')$. A consistent heuristic is also admissible: the value $h(n)$ is at most the true cost from n to the goal state.