Notes
Seminar on Machine Learning and Neural Networks

Chapter 1

Linear Regression

The goal of regression is to construct a model around some given data, so that new values can be predicted with a known measure of certainty. For example, predicting the value of the stock of a given company given its history or predicting blood sugar levels acording to some prior data about a patient's blood and food intake.

1.1 A first example: fitting with polynomials

A first approach to regression is modeling data by fitting. Say we have N tuples of data $\mathscr{D} = \{(x_1, t_1), \dots, (x_N, t_N)\}$ and we want to predict, given a new $\widetilde{x}$, a possible value for $\widetilde{t}$. One way to do it is to **fit** a polynomial through the points in $\mathscr{D}$ (and by that we mean to find a polynomial that passes near t_j at x_j). We fix, then, an approximation function (also known as a predictor) of the form

$$y(x, w) = w_0 + w_1 x + \cdots + w_m x^m = \sum_{i=1}^{m} w_i x^i$$

we call $w = (w_0, \dots, w_m)$ the **parameters**, and they're the freedom we have. They're, to put it in CGP Gray's words, a set of dials we need to tune in order to approximate $y(x, w)$ to the points in $\mathscr{D}$ as much as we can.

A first approximation to finding w is done by minimizing the distance between $y(x_j, w)$ and t_j. Mathematically speaking, we want to minimize the **sum-of-squares error function**

$$E(w) = \frac{1}{2} \sum_{j=1}^{N} (y(x_j, w) - t_j)^2$$

(which is completely equivalent to minimizing the euclidean distance between $y(x_j, w)$ and t_j). Computing $\nabla E = 0$ renders a system of $m+1$ linear equations to be solved: for $k \in \{0, 1, \dots, m\}$

$$\begin{aligned}
\frac{\partial}{\partial w_k} E(w) &= \frac{1}{2} \sum_{j=1}^{N} \frac{\partial}{\partial w_k} \left(\sum_{i=1}^{m} w_i x_j^i - t_j \right)^2 \\
&= \sum_{j=1}^{N} \left(\sum_{i=0}^{m} w_i x_j^i - t_j \right) (x_j^k) \\
&= \sum_{j=1}^{N} \left(\sum_{i=0}^{m} w_i x_j^{i+k} - t_j x_j^k \right) \\
&= \sum_{i=0}^{m} \left(\sum_{j=1}^{N} x_j^{i+k} \right) w_i - \sum_{j=1}^{N} t_j x_j^k
\end{aligned}$$

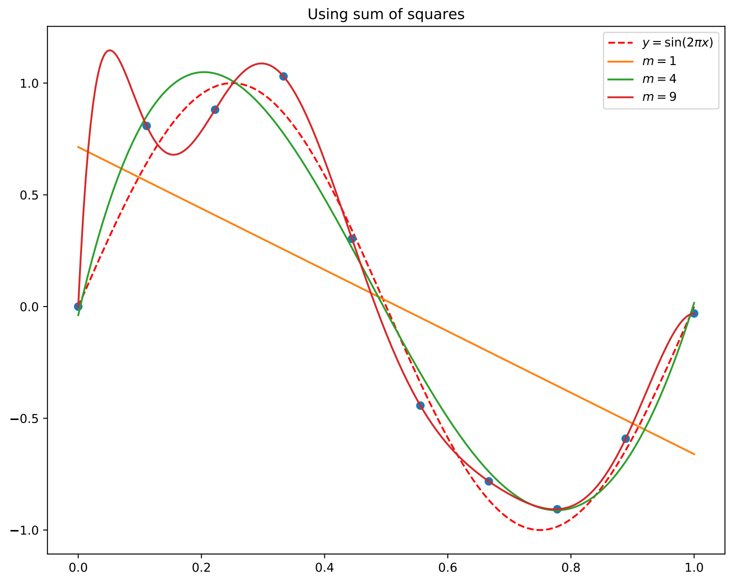

Figure 1.1: An example of polynomial fitting for $\sin(x)+\epsilon$ for a random noise ϵ.

that is, the system to be solved is

$$\sum_{i=0}^{m}\left(\sum_{j=1}^{N} x_j^{i+k}\right) w_i = \sum_{j=1}^{N} t_j x_j^k, \;\; k \in \{0,\dots,m\}$$

Here is a function in python that outputs these coefficients by solving the linear system:

```
import numpy as np
from scipy import linalg

def poly_fit(x, t, m):
    A = np.zeros((m + 1, m + 1))
    for k in range(m + 1):
        for i in range(m + 1):
            A[k, i] = np.sum(x ** (i + k))
    b = np.array([np.sum(t * x**k) for k in range(m + 1)])
    w = linalg.solve(A, b)
    return w
```

Consider this example: say we want to fit a set $\{(x_1,t_1),\dots,(x_{10},t_{10})\}$ that come from the function $y = \sin(2\pi x)$ plus some noise that's distributed normally with mean 0 and variance 0.1. After implementing this algorithm, the aproximations for $M = 1,4,9$ can be found in figure 1.1

At $m = 1$ we're barely approximating the points, at $m = 4$ we're approximating $\sin(x)$ rather well and at $m = 9$ the parameters become so finely tuned that the polynomial passes almost exactly through the points, but at the cost of enormous oscilations near the endpoints. We see, then, that if we assume that the points come from some sort of pattern ($y = \sin(x)$ in this case), using a lot of parameters works against us. This is known as **overfitting**.

1.2 Maximizing the likelihood and the posterior

In order to prevent overfitting, we will take a more Bayesian approach. Going Bayesian also works in the spirit of modeling how **certain** we are on the results of our models.

Let's remember Bayes' theorem[1]. It follows naturally from the definition of conditional probability that

$$p(X)p(Y|X) = p(X,Y) = p(Y)p(X|Y)$$

which can be rewritten as

$$p(Y|X) = \frac{p(X|Y)p(Y)}{p(X)}$$

If we consider $p(X|Y)$ as the **likelihood function** that outputs how probable is X after choosing Y, Bayes' theorem tell us that the **posterior probability** $p(Y|X)$ is a normalization of the product between the **prior probability** $p(Y)$ and the **likelihood function** $p(X|Y)$.

Back to the problem of curve fitting, having the data $\mathscr{D} = \{(x_1, t_1), \dots, (x_N, t_N)\}$ and calling $t = (t_1, \dots, t_N)^T$, we are interested in finding w such that $p(t|w)$ is maximal, that is, such that it is quite probable that t comes from the parameters w in an approximation that somehow involves $y(x, w)$. For (as Bishop calls it) analytic simplicity, we model the probability (i.e. certainty) that t comes from the hidden pattern in the x_js with a normal distribution:

$$\begin{aligned} p(t|w, x, \beta) &= \mathcal{N}(t|y(x, w), \beta^{-1}) \\ &= \frac{1}{(2\pi\sigma^2)^{1/2}} \exp\left(-\frac{1}{2\sigma^2}(t - y(x, w))^2\right) \end{aligned}$$

where $\beta = 1/\sigma^2$ is called the **precision**, σ^2 is the variance and the mean, in this case, is $y(x, w)$. As we're varying w and β in $p(t|w, x, \beta)$, it can be interpreted as a likelihood function of these parameters and precision. Because we want to maximize the probability that t comes from these variables, we call this method **maximum likelihood**. We find w and β by minimizing $-\ln(p)$ (which is equivalent to maximizing p itself, because ln is monotonically increasing):

$$-\ln(p(t|w, x, \beta) = \frac{\beta}{2}\sum_{j=1}^{N}(y(x_j, w) - t_j)^2 - \frac{N}{2}\ln(\beta) + \frac{N}{2}\ln(2\pi)$$

Note that by changing the perspective on the behavior of the data (i.e. supposing t is normally distributed around the fitting polynomial $y(x, w)$) the sum-of-squares error function naturally arises. This calls for the definition of an **error function** to be precisely the negative of the logarithm of the likelihood function.

We can add another assumption in order to prevent overfitting: that the parameters have a prior distribution $p(w|\alpha)$ (here α is another parameter to be determined). For example, consider a multivariate normal distribution

$$p(w|\alpha) = \mathcal{N}(w|0, \alpha^{-1}I) = \left(\frac{\alpha}{2\pi}\right)^{(M+1)/2} \exp\left\{-\frac{\alpha}{2}w^T w\right\}$$

We can then **maximize the posterior** by using Bayes' theorem:

$$p(w|t, x, \alpha, \beta) \propto p(t|x, \beta)p(w|\alpha)$$

If we consider the error function associated with this probabilty, we get:

[1] For a good summary of the basic definitions in probability needed for this chapter and the rest of this notes, take a look at appendix []

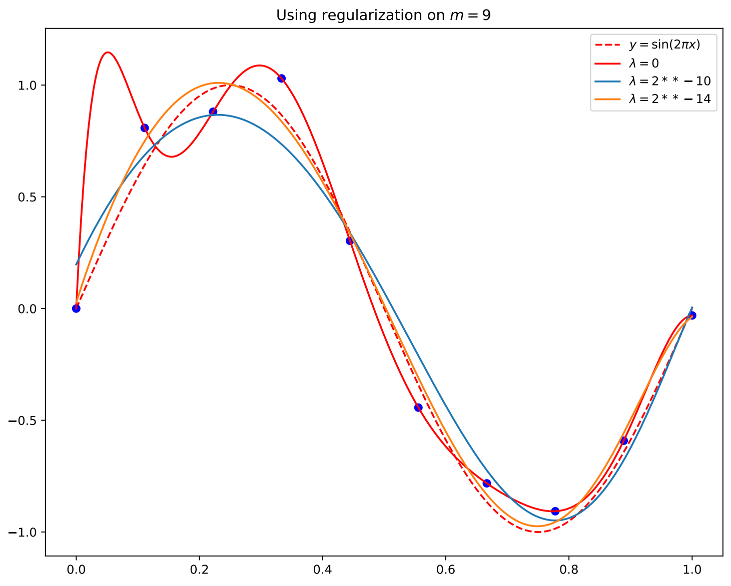

Figure 1.2: Regularization of sum-of-squares with multiple values for λ.

$$-\ln(p(w|t,x,\alpha,\beta)) = \frac{\beta}{2}\sum_{j=1}^{N}(y(x_j,w)-t_j)^2 + \frac{\alpha}{2}w^Tw - \frac{N}{2}\ln(\beta) - \frac{M+1}{2}\ln(\alpha) + C$$

For some constant C. Focusing on the part that concerns w we get the **regularized sum-of-squares function**:

$$E_{\text{MAP}}(w) = \frac{\beta}{2}\sum_{j=1}^{N}(y(x_j,w)-t_j)^2 + \frac{\alpha}{2}w^Tw$$

By taking into account the quadratic term on the parameters, we're forcing the system to avoid overfitting, which usually occurs when the parameter get big in size. The normal equations of such a system is only modified by the addition of a *diagonal* term. One usually modifies $E_{\text{MAP}}(w)$ by dividing it by β and calling $\lambda = \alpha/\beta$. This parameter λ measures the relevance of this regularizing parameter.

1.3 Linear models of regression

This past example (fitting a set of data with a polynomial) is a particular example of **linear regression models**. The main characteristic of this models is that they behave linearly on the parameters. Generally speaking, we define a linear regression model by an approximation function of the form

$$y(x,w) = w^T\phi(x) = w_0 + \sum_{j=1}^{M-1} w_j\phi_j(x)$$

where $w = (w_0,\ldots,w_{M-1})^T$ are the parameters and $\phi(x) = (1,\phi_1(x),\ldots,\phi_{M-1}(x))^T$ are called the **basis functions**. In such a model, the parameters are a linear combination in which the coefficients are functions of x. In our

particular example (polynomial fitting), the basis functions could be $\phi_j(x) = x^j$, but we have the freedom to use any ϕ we want and introduce non-linearity in the input values.
We follow the same schematics as in the example: we model the distribution of our data $\mathscr{D} = \{(x_j, t_j)\}_{j=1}^N$ with a normal distribution whose mean is $y(x, w)$:

$$p(t|x, w, \beta) = \mathcal{N}(t|y(x, w), \beta^{-1})$$

Just as we discussed, the error function that we get from this assumption is the sum-of-squares:

$$E(w) = \frac{1}{2}\sum_{j=1}^{N}(t_j - w^T\phi(x_j))^2$$

and when we minimize this function, we get that the solution is given by

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

where by t we denote the vector $t = (t_1, \dots, t_N)^T$ and Φ is the following matrix (called the **design matrix**):

$$\Phi = \begin{bmatrix} \phi_0(x_1) & \phi_1(x_1) & \cdots & \phi_{M-1}(x_1) \\ \phi_0(x_2) & \phi_1(x_2) & \cdots & \phi_{M-1}(x_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_0(x_N) & \phi_1(x_N) & \cdots & \phi_{M-1}(x_N) \end{bmatrix}_{N\times M}$$

The quantity $(\Phi^T\Phi)^{-1}\Phi^T$ is fairly common in our discussions, so it receives a special name: the **Moore-Penrose pseudoinverse**. Note that we could also consider a regularized version of the error function (as when maximizing the posterior in the example), so that the error function becomes

$$E(w) = \frac{1}{2}\sum_{j=1}^{N}(t_j - w^T\phi(x_j))^2 + \frac{\lambda}{2}w^T w$$

When we minimize this error function we get a slight variation of the pseudoinverse solution:

$$w^*_{MAP} = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T t$$

Recall that we got w^* by solving $\nabla E(w) = 0$ analytically, but we can estimate w^* by **sequential learning**, that is, using gradient descent to arrive to a minimum by the following iterative scheme:

$$w^{(\tau+1)} = w^\tau - \eta\nabla E_j$$

with w^0 randomly initialized. η, the **learning parameter**, needs to be selected carefully taking into account the following trade-off: if it's too big, the scheme diverges, if it's too small, the scheme takes a longer time to converge. This is a common approach when it comes to applications in which the data is flowing *in real time* (e.g. a web application), because E_j represents the error term $(t_j - y(x_j, w^\tau))^2$ coming from the j-th data pair.

1.4 Summary of Linear Regression

The model: $y(x, w) = w^T\phi(x)$ where $x \in \mathbb{R}^d$ is an input vector, $w = (w_0, \dots, w_{M-1})^T$ is the vector of parameters and $\phi(x) = (1, \phi_1(x), \dots, \phi_{M-1}(x))^T$ are called basis functions.

The learning: Given a data set $\mathscr{D} = \{(x_j, t_j)\}_{j=1}^N$, the parameters w can be learned by minimizing the error function $E(w) = \frac{1}{2}\sum_{j=1}^N (t_j - w^T\phi(x_j))^2 + \frac{\lambda}{2} w^T w$. There are two ways to solve for $\nabla E(w)$

- **Analitically**, which renders the following solution: $w^*_{MAP} = (\Phi^T\Phi + \lambda I)^{-1}\Phi^T t$.
- **Sequentially**, in which we gradient descent in every error term $E_j = (y(x_j, w) - t_j)^2$:

$$w^{(\tau+1)} = w^{(\tau)} - \eta\nabla E_j(w^{(\tau)}).$$

$w^{(0)}$ can be initialized randomly.

Its meaning: The sum-of-squares error term comes from the assumption that $p(t) = \mathcal{N}(t|y(x, w), \beta^{-1})$. The regularization term comes from maximizing the posterior after supposing that $p(w) = \mathcal{N}(w|0, \alpha^{-1})$, and λ is defined as α/β.

Chapter 2

Linear Classification

Classification problems consist on constructing a model that takes every x in the input space and maps it to a discrete set $\{C_1, \dots, C_K\}$ of *classes*, given that we already have a set of classified data $\mathscr{D} = \{(x_i, t_i)\}_{i=1}^N$. There are plenty of examples of real life problems that require a classification, for example

- We have diagnostic images of a liver, and we want to determine wheter a new patient has liver cancer or not.
- We have a collection of labeled handwritten digits (such as the MNIST database), and we want to determine which digit appears in a new image.
- We have a song, and we want to determine whether it is a happy, upbeat one or a melancholic one.
- A bank may have plenty of information about their clients, and may want to construct a model to determine if a new client will default on their credit or not.
- A Nature article explains how a model for classifying skin lesions at dermatologist-level was constructed.

Many of these examples may require heavy machinery, but for now we will introduce the models of classification by considering the most basic ones: linear models of classification.
There are three ways to construct a model, and they go (in increasing level of complexity) like this:

(a) Model a **discriminant function** $y(x, w)$ that classifies x.

(b) Model $P(C_k|x)$ for $k \in \{1, \dots, K\}$ and use this information to take decisions over x, such as classifying x to the class C_k such that $P(C_k|x) \geq P(C_j|x)$ for all $j \in \{1, \dots, K\}$. Such models are called **Probabilistic Discriminant Functions**, because the $y_k(x) = P(C_k|x)$ now serve as discriminant functions.

(c) Model $P(x|C_k)$ and $P(C_k)$ (i.e. modeling the joint distribution $P(x, C_k)$) and then use Bayes to get $P(C_k|x)$. These models are called **Generative Models**, because we can marginalize the joint distribution for the x and generate new training data.

2.1 Linear Discriminant Models

Our first approach to classification will be using linear discriminant models, we will learn the parameters of a series of functions of the form

$$y_k(x, w_k) = w_k^T x + w_{k0}, \; k \in \{1, \dots, K\}$$

such that we classify x to C_k if $y_k(x, w_k) > y_j(x, w_j)$ for all $j \neq k$. Naming names, we call $w_k = (w_{k1}, \dots, w_{kK})^T$ the parameters, w_{k0} is a special parameter called the **bias**, and $-w_{k0}$ is called the **threshold**. We can get all these linear combinations together into a vector and consider only one discriminant function

$$y(x, w) = \begin{bmatrix} y_1(x, w_1) \\ y_2(x, w_2) \\ \vdots \\ y_K(x, w_K) \end{bmatrix} = W^T x + W_0$$

where W is a matrix whose columns are precisely $w_1, w_2, \dots, w_K$, and $W_0 = (w_{1,0}, w_{2,0}, \dots, w_{K,0})^T$. Such a model is usually called a 1-of-K scheme.

2.1.1 The geometry of this model

Let's consider the geometry that such a scheme induces. Call the regions $R_k := \{x \mid x \text{ is classified to } C_k\} \subseteq \text{Inp}$ and ∂R_k the **decision boundaries**. We claim that these decision boundaries are hyperplanes and that the R_ks are simply connected and convex.

Indeed, the first claim follows from the fact that the decision boundary between R_i and R_j is given by $\{x \mid y_i(x) = y_j(x)\}$, because then

$$\begin{aligned} & y_i(x) = y_j(x) \\ & \quad \Rightarrow w_i^T x + w_{i0} = w_j^T x + w_{j0} \\ & \quad \Rightarrow (w_i - w_j)^T x + (w_{i0} - w_{j0}) = 0 \end{aligned}$$

thus, the decision boundary is given by a hyperplane. Now let's prove that each region is convex and simply connected: let $x, x' \in R_k$, consider then a convex combination $\hat{x} = \lambda x + (1-\lambda)x'$, we need to prove that for every $\lambda \in [0,1]$, $\hat{x} \in R_k$, i.e., $y_k(\hat{x}) > y_j(\hat{x})$ for every $j \neq k$. Indeed,

$$\begin{aligned} y_k(\hat{x}) &= y_k(\lambda x + (1-\lambda)x') \\ &= w_k^T(\lambda x + (1-\lambda)x') + w_{k0} \\ &= \lambda(w_k^T x + w_{k0}) + (1-\lambda)(w_k^T x' + w_{k0}) \\ &= \lambda y_k(x) + (1-\lambda) y_k(x') \\ &> \lambda y_j(x) + (1-\lambda) y_j(x') \\ &= y_j(\hat{x}) \end{aligned}$$

(notice that we skipped the same algebraic process between the last steps). This proves that the model we're dealing with splits the regions into convex polytopes. This is a rigid geometry!, we're not to expect that the actual pattern behind the data be **linearly separable**. This is not generally the case, and we require more sophisticated methods in order to classify data that's not very linearly separable.

2.1.2 Learning the parameters

We now have a linear model, now let's talk about a couple ways we can *learn* the parameters from training data.

Sum-of-squares

We can draw an analogy with regression, and try to fit a function using least squares. We could define an error associated with the training data $\mathcal{D} = \{(x_i, t_i)\}_{i=1}^N$:

$$E_{\mathscr{D}}(w_1, \dots, w_k) = \frac{1}{2} \sum_{k=1}^{K} \sum_{n=1}^{N} (x_n^T w_k + w_{k0} - t_{nk})^2$$

in order to get the minima of this error, we can rewrite it as the trace of a matrix. Let

$$\widetilde{X} = \begin{bmatrix} 1, x_1^T \\ \cdots \\ 1, x_N^T \end{bmatrix}_{N \times (D+1)} = \begin{bmatrix} \widetilde{x}_1^T \\ \cdots \\ \widetilde{x}_N^T \end{bmatrix}_{N \times (D+1)}$$

$$\widetilde{W} = \left[\widetilde{w}_1 \mid \cdots \mid \widetilde{w}_K \right]_{(D+1) \times K}$$

where $\widetilde{w}_k = (w_{k0}, w_{k1}, \dots, w_{kK})^T = (w_{k0}, w_k^T)^T$, and finally

$$T = \begin{bmatrix} t_1^T \\ \cdots \\ t_N^T \end{bmatrix}_{N \times K}$$

Then the error above defined can be written as

$$E_{\mathscr{D}}\left(\widetilde{W}\right) = \frac{1}{2} \operatorname{Tr}\left((\widetilde{X}\widetilde{W} - T)^T (\widetilde{X}\widetilde{W} - T) \right)$$

we can differentiate with respect to $\widetilde{W}$ in order to get the minimum, which is given by

$$\widetilde{W}^* = (\widetilde{X}^T \widetilde{X})^{-1} \widetilde{X}^T T$$

Notice that this is the same solution as we got in the linear regression scenario, and that the Moore-Penrose pseudoinverse pops up once more.

The perceptron algorithm

[to be filled]

2.2 Probabilistic Discriminative Models

Chapter 3

Feed-Forward Neural Networks

When trying to model the functional relation between values in a dataset $\mathscr{D} = \{(\vec{x}_n, \vec{t}_n)\}_{n=1}^{N}$, we have tried to fit an approximating function of the form

$$\vec{y}(\vec{x}; w) = \sum_{i=0}^{M-1} \phi_i(\vec{x}) w_i$$

where $\phi(\vec{x}) = (1, \phi_1(\vec{x}), \ldots, \phi_{M-1}(\vec{x}))$ are basis functions defined *a priori* by the modeler. In classification, for example, it was essential to select ϕ in such a way that the set $\widetilde{\mathscr{D}} = \{(\phi(\vec{x}_n), \vec{t}_n)\}_{n=1}^{N}$ were linearly separable (which is, indeed, asking too much).

The idea of (Artificial) Feed-Forward Neural Networks is to add more flexibility to the model by turning these basis functions into a non-linear transformation of a linear combination of the input variables with parameters. That is, the model in this case looks like this:

$$\vec{y}(x; w) = \sigma_d \left(W_{(d)}^T \sigma_{d-1}(W_{(d-1)}^T \cdots \sigma_1(W_{(1)}^T \vec{x} + \vec{b}_{(1)}) + \vec{b}_{(2)}) \cdots + \vec{b}_{(d)} \right) \tag{3.1}$$

Where $w = (W_{(t)}, \vec{b}_t)_{t=1}^{d}$ are matrices and vectors with parameters and $\sigma_1, \ldots, \sigma_d$ are called **activation functions**. We call d the amount of layers in the neural network. We can represent this fact visually using **computation graphs** (our local name, it seems like it isn't a generalized term).

3.1 Computation graphs

We will end up representing our model using a directed acyclical graph. In this graph, the edges will represent our parameters and the nodes will represent values that are computed in the following way: the value in node v is given by

$$v = \sigma \left(\sum_{u \to v} w(u, v) \cdot u \right)$$

[figure1]

Using this language, the model (3.1) can be described by a directed acyclical d-partite graph, in which the connections between the d parts are complete. This graph can be found in figure [figure2].

[figure2]