

---

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Supervised learning . . . . .	3
1.2	Unsupervised learning . . . . .	3
1.3	Reinforcement learning . . . . .	3
<b>2</b>	<b>Classifier</b>	<b>5</b>
2.1	Linear classifiers . . . . .	6
2.2	Features . . . . .	8
<b>3</b>	<b>Logistic Regression</b>	<b>11</b>
3.1	Linear logistic classifier . . . . .	11
3.2	Gradient descent . . . . .	12
3.3	Linear regression . . . . .	14
3.4	Regularizers . . . . .	15
<b>4</b>	<b>Neural Network</b>	<b>17</b>
4.1	Introduction . . . . .	17
4.2	Training . . . . .	18



---

# Chapter 1

## Introduction

Some kinds of learning include;

### 1.1 Supervised learning

Given a dataset of pair

$$\mathcal{D}_n = \{(x^{(1)}, y^{(1)}), \dots, (x^{(n)}, y^{(n)})\} \quad (1.1)$$

we wish to establish a relationship between  $x^{(i)}$  and  $y^{(i)}$ . Typically,  $x^{(i)} \in \mathbb{R}^d$  is a representation of input, called **feature representation**. Based on the format of the output we can have different types of supervised learning:

**Classification** when the set of possible values of  $y^{(i)}$  is discrete (small finite set). If there two possible values then the classification problem is *binary* otherwise, it is called *multi-class*.

**Regression** when the set of possible values of  $y^{(i)}$  is continuous (or a large finite set). That is,  $y^{(i)} \in \mathbb{R}^k$ .

### 1.2 Unsupervised learning

Given a dataset we wish to find some patterns or structures in it. There are several types of unsupervised learning:

**Density estimation** The data is i.i.d from some distribution  $p_X(x)$ . The goal is to predict the probability  $p_X(x^{(n+1)})$ .

**Clustering** the goal is to find a partitioning of the sample data that groups together samples that are similar. Clustering is sometimes used in density estimation.

**Dimensionality reduction** the goal is to re-represent the same data in  $\mathbb{R}^l$  where  $l < d$ .

### 1.3 Reinforcement learning

The goal is to learn a mapping from input values to output values without a direct supervision signal. There is no training set specified *a priori*. Instead, the learning problem is framed as an agent interacting with an environment. Looking at input as our states and output as a

transition between states, we can assign a reward value  $r^{(i,j)}$  to each such transition. We aim to find a policy  $\pi$  that maximizes the long-term sum or average of rewards.

---

# Chapter 2

## Classifier

To predict, we come up with a **hypothesis**. A hypothesis is a parametrized function that maps input to output

$$y = h(x; \theta), \quad h \in \mathcal{H}, \theta \in \Theta$$

where  $\mathcal{H}$  is our *hypothesis class*. We wish to find the parameters  $\theta$  that matches our data well. One way to evaluate how well our hypothesis predicts is to introduce a **loss function** (or *cost function*),  $L(a, a_h)$  where  $a, a_h$  are in the output set and the loss function assigns a value to how close our prediction  $a_h$  when the actual value is  $a$ . We wish that our hypothesis to have the least loss on new data.

$$\mathcal{E}(h) = \frac{1}{n'} \sum_{i=n+1}^{n+n'} L(h(x^{(i)}; \theta), y^{(i)})$$

One way to do this is minimize the training error

$$\hat{\mathcal{E}}(h) = \frac{1}{n} \sum_{i=1}^n L(h(x^{(i)}; \theta), y^{(i)})$$

There are several types of loss function

### 0-1 Loss

$$L(a, a_h) = \begin{cases} 0 & \text{if } a = a_h \\ 1 & \text{otherwise} \end{cases}$$

### Squared loss

$$L(a, a_h) = (a - a_h)^2$$

### Linear loss

$$L(a, a_h) = |a - a_h|$$

**Asymmetric loss** For example, maybe guessing negative wrong is costlier than guessing positive wrongly, in a binary classification problem.

The model we use, typically, selects  $h$  and we need to minimize the loss (or any other optimization) on the  $\theta$  so that our prediction *fits* the data. To determine a good  $\theta$  we need algorithms, *learning algorithms*. Given a classifier  $h$  we can easily evaluate its performance by testing it on new data. However, to evaluate a learning algorithm we have to first train it on some data and then evaluate the resulting classifier on the testing data. Doing this multiple

gives us an estimate of how well the algorithm works. In most cases, we do not have access to a lot of data (we don't know the distribution). In these cases we can re-use data using cross validation

---

**Algorithm 1:** cross\_validate ( $\mathcal{D}, k$ )

---

```

divide  $\mathcal{D}$  into  $k$  equally sized chunks  $\mathcal{D}_1, \dots, \mathcal{D}_k$ 
for  $i = 1 \rightarrow k$  do
    train  $h_i$  on  $\mathcal{D} - \mathcal{D}_i$ 
    compute  $\mathcal{E}_i(h_i)$  on the test data  $\mathcal{D}_i$ 
return  $\frac{1}{k} \sum_{i=1}^k \mathcal{E}_i(h_i)$ 

```

---

## 2.1 Linear classifiers

A linear classifier has the following form

$$h(x; \theta, \theta_0) = \text{sign}(\theta^T x + \theta_0) \quad \theta \in \mathbb{R}^d, \theta_0 \in \mathbb{R}$$

### 2.1.1 Random linear classifier

One way to select the parameters  $\theta$  and  $\theta_0$  is to randomly select them and return the best one.

---

**Algorithm 2:** random\_linear\_classifier ( $\mathcal{D}_n, k$ )

---

```

for  $j = 1 \rightarrow k$  do
     $\theta^{(j)} = \text{Random}(\mathbb{R}^d)$ 
     $\theta_0^{(j)} = \text{Random}(\mathbb{R})$ 
 $j^* = \text{argmin}_{1 \leq j \leq k} \hat{\mathcal{E}}\left(h\left(x, \theta^{(j)}, \theta_0^{(j)}\right)\right)$ 
return  $(\theta^{(j^*)}, \theta_0^{(j^*)})$ 

```

---

### 2.1.2 Perceptron

A more intelligent way of finding the parameters is to update when we encounter a mistake. The simplest update rule is the **perceptron**, which is as follows

$$\theta', \theta'_0 \leftarrow \theta + y^{(i)} x^{(i)}, \theta_0 + y^{(i)}$$

This update increases the magnitude of  $y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0)$  as shown below, and hence in enough iterations, the sign will become positive.

$$\begin{aligned}
 y^{(i)} \cdot (\theta'^T x^{(i)} + \theta'_0) &= y^{(i)} (\theta + y^{(i)} x^{(i)})^T x^{(i)} + y^{(i)} (\theta_0 + y^{(i)}) \\
 &= y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0) + (y^{(i)})^2 (\|x^{(i)}\|^2 + 1) \\
 &= y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0) + \|x^{(i)}\|^2 + 1
 \end{aligned}$$

however it is not clear how other mistakes will affect the current guess.

---

**Algorithm 3:** perceptron  $(\mathcal{D}_n, T)$ 

---

```

 $\theta = 0$ 
 $\theta_0 = 0$ 
for  $t = 1 \rightarrow T$  do
  for  $i = 1 \rightarrow n$  do
    if  $y^{(i)}(\theta^T x^{(i)}) + \theta_0 \leq 0$  then
       $\theta = \theta + y^{(i)} x^{(i)}$ 
       $\theta_0 = \theta_0 + y^{(i)}$ 
return  $(\theta, \theta_0)$ 

```

---

By adding another dimension to our data set we can simplify our prediction to pass through the origin

$$x' = \begin{bmatrix} x_1 & \dots & x_n & 1 \end{bmatrix}, \quad \theta' = \begin{bmatrix} \theta & \theta_0 \end{bmatrix}$$

$$\implies \theta'^T x' = \theta^T x + \theta_0$$

Therefore, one can also simplify the Line 3 to the following

---

**Algorithm 4:** perceptron  $(\mathcal{D}_n, T)$ 

---

```

 $\theta = 0$ 
for  $t = 1 \rightarrow T$  do
  for  $i = 1 \rightarrow n$  do
    if  $y^{(i)}(\theta^T x^{(i)}) + \theta_0 \leq 0$  then
       $\theta = \theta + y^{(i)} x^{(i)}$ 
return  $\theta$ 

```

---

To examine the convergence of the perceptron algorithm we need the following definitions. A dataset  $\mathcal{D}_n$  is **linearly separable -through the origin** if there is some  $\theta$  such that

$$y^{(i)} \theta^T x^{(i)} > 0 \quad \forall i$$

and similarly it is linearly separable if there are some  $\theta, \theta_0$  such that

$$y^{(i)} \cdot (\theta^T x^{(i)} + \theta_0) > 0 \quad \forall i$$

The **margin** of a labeled data point  $(x, y)$  with respect to a separator (hyperplane)  $\theta, \theta_0$  is

$$y \cdot \frac{\theta^T x + \theta_0}{\|\theta\|}$$

which basically quantifies how well  $\theta$  approximates the data point  $(x, y)$  in a data set  $\mathcal{D}_n$ . Also the margin of  $\mathcal{D}_n$  w.r.t  $\theta, \theta_0$  is the minimum of all margins:

$$\min_i y^{(i)} \cdot \frac{\theta^T x^{(i)} + \theta_0}{\|\theta\|}$$

**Theorem 2.1 (Perceptron convergence theorem).** *If there exists a vector  $\theta^*$  such that the margin of database with respect to  $\theta^*$  is greater than  $\gamma > 0$  and then norm  $\|x^{(i)}\| \leq R$  for some  $R$  then perceptron will make at most  $\left(\frac{R}{\gamma}\right)^2$  updates/mistakes.*

*Proof.* The idea of the proof is to put an increasing lower bound on the cosine of the angle between the  $k_{\text{th}}$  update  $\theta^{(k)}$  and  $\theta^*$ . Note that

$$\cos(\angle \theta^*, \theta^{(k)}) = \frac{(\theta^*)^T \theta^{(k)}}{\|\theta^*\| \|\theta^{(k)}\|}$$

we know that for some  $i$ ,  $\theta^{(k)} = \theta^{(k-1)} + y^{(i)}x^{(i)}$ , then

$$\begin{aligned} (\theta^*)^T \theta^{(k)} &= (\theta^*)^T (\theta^{(k-1)} + y^{(i)}x^{(i)}) \\ &= (\theta^*)^T \theta^{(k-1)} + y^{(i)}(\theta^*)^T x^{(i)} \\ &\geq (\theta^*)^T \theta^{(k-1)} + \gamma \|\theta^*\| \\ &\geq k\gamma \|\theta^*\| \end{aligned}$$

also

$$\begin{aligned} \|\theta^{(k)}\|^2 &= (\theta^{(k-1)} + y^{(i)}x^{(i)})^T (\theta^{(k-1)} + y^{(i)}x^{(i)}) \\ &= \|\theta^{(k-1)}\|^2 + 2y^{(i)}(\theta^{(k-1)})^T x^{(i)} + (y^{(i)})^2 \|x^{(i)}\|^2 \\ &\leq \|\theta^{(k-1)}\|^2 + \|x^{(i)}\|^2 \\ &\leq \|\theta^{(k-1)}\|^2 + R^2 \\ &= kR^2 \end{aligned}$$

therefore

$$\cos(\angle \theta^*, \theta^{(k)}) \geq \sqrt{k} \frac{\gamma}{R}$$

Since the cosine can not exceed one therefore the we at most make

$$k \leq \left(\frac{R}{\gamma}\right)^2$$

mistakes. ■

## 2.2 Features

### 2.2.1 Transformation

As we saw we can transform linearly separable dataset to another linearly separable dataset but without an offset. What happens if the original dataset is not linearly separable? For example, *xor dataset*:

$$\mathcal{D} = \{((-1, -1), 1), ((-1, 1), -1), ((1, -1), -1), ((1, 1), 1)\}$$

is not linearly separable in 2 dimensions. A transformation that might be applicable here is **polynomial basis**. A polynomial basis transformation of order  $k$ , transforms a feature  $x \in \mathbb{R}^d$  to

$$\phi(x) = (x_1^{\alpha_1} \dots x_d^{\alpha_d}), \quad \sum_{i=1}^d \alpha_i \leq k, \forall i, \alpha_i \geq 0$$

which has  $\binom{k+d}{d}$  dimension.

For example, the transformation  $\phi(x_1, x_2) = (x_1, x_2, x_1x_2)$  makes the xor dataset linearly separable-through the origin.



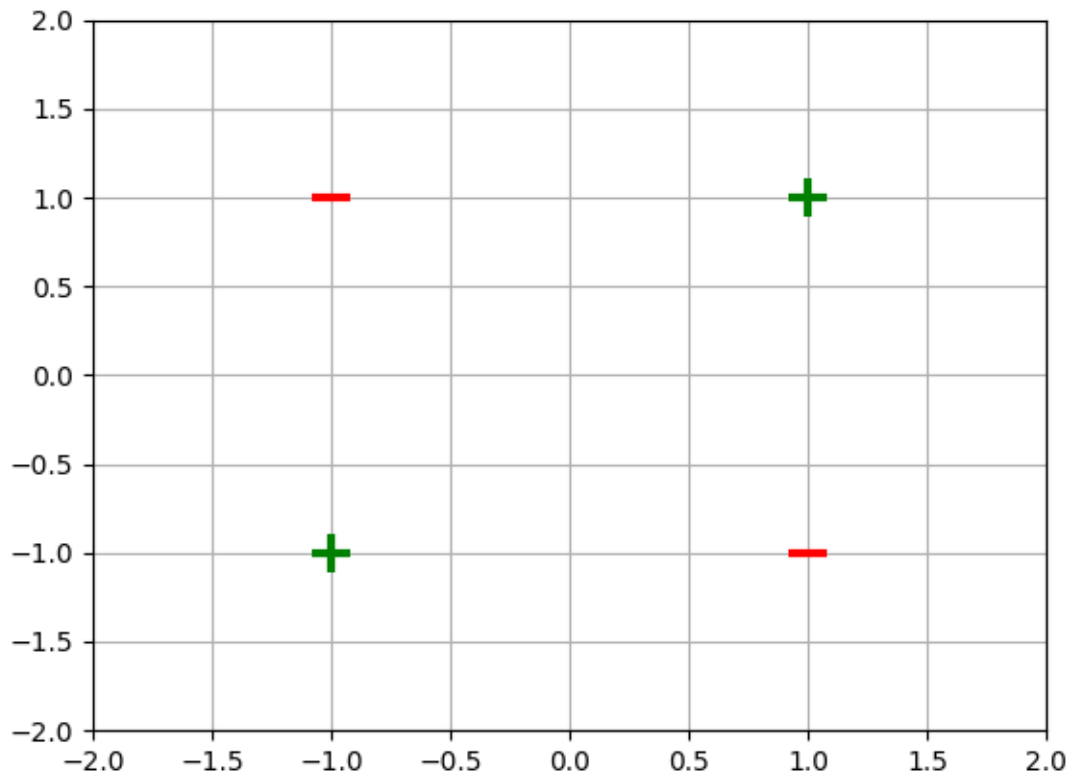


Figure 2.1: XOR dataset

### 2.2.2 Representation

we can represent a discrete feature as

1. numeric
2. thermometer code (a vector of  $m$  booleans where  $1 \dots j$  bits are on and the rest are off)
3. one-hot (a vector of  $m$  booleans where  $j_{\text{th}}$  bit is on and the rest are off)
4. factoring (group information of a feature based on its structure maybe)

For numeric feature we would like to standardize as follows

$$\tilde{x}_j = \frac{x_j - \bar{x}_j}{\sigma}$$

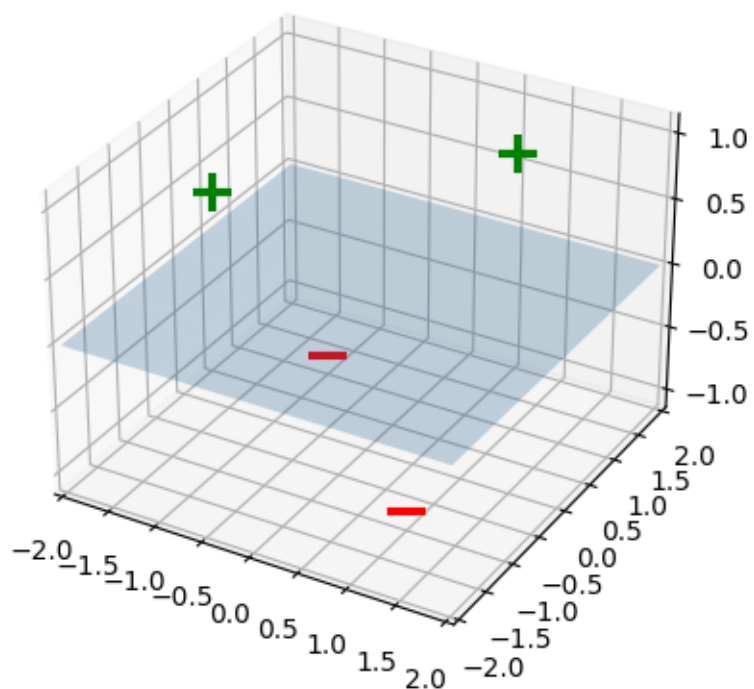


Figure 2.2: Transformed XOR dataset

---

# Chapter 3

## Logistic Regression

In machine learning we wish to optimize a function like  $J(\theta)$ . Usually a function in form

$$\begin{aligned} J(\theta) &= \left( \frac{1}{n} \sum_{i=1}^n L(h(x^{(i)}; \theta), y^{(i)}) \right) + \lambda R(\theta) \\ &= \mathcal{E}_n + \lambda R(\theta) \end{aligned}$$

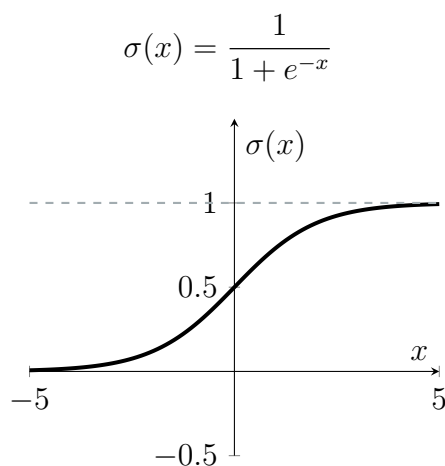
where  $R$  is the **regularization function** and  $\lambda$  is a hyperparameter. A common regularizer is

$$R(\theta) = \|\theta - \theta_{\text{prior}}\|^2$$

where  $\theta_{\text{prior}}$  is the value that we want  $\theta$  to be close to.

### 3.1 Linear logistic classifier

The problem of minimizing 0-1 Loss problem is NP-hard. A problem with sign is that incremental change are hard to find because of the discrete nature of the function hence, to smooth out the sign function we use sigmoid function



Equivalently, we want to make classifier that predict +1 when  $\sigma(\theta^T x + \theta_0) > p = 0.5$  and -1 otherwise. The value  $p$  is called the **prediction threshold**.

Loss on all data is inversely related to the probability that  $(\theta, \theta_0)$  assigns to the data. Assuming that points in data set are independent.

$$g^{(i)} = \sigma(\theta^T x + \theta_0)$$

$$p^{(i)} = \begin{cases} g^{(i)} & \text{if } y^{(i)} = 1 \\ 1 - g^{(i)} & \text{if } y^{(i)} = 0 \end{cases}$$

and we wish to maximize the probability ( $\Theta$  here means  $(\theta, \theta_0)$ ):

$$p(\Theta; \mathcal{D}_n) = \prod_{i=1}^n p^{(i)} = \prod_{i=1}^n (g^{(i)})^{y^{(i)}} (1 - g^{(i)})^{(1-y^{(i)})}$$

Using the log-likelihood

$$\Rightarrow \mathcal{L}_{LL}(p) = \sum_{i=1}^n \mathcal{L}_{LL}(g^{(i)}, y^{(i)}) = \sum_{i=1}^n y^{(i)} \ln(g^{(i)}) + (1 - y^{(i)}) \ln(1 - g^{(i)})$$

then our loss function would be

$$L(h(x^{(i)}, \Theta), y^{(i)}) = -\mathcal{L}_{LL}(g^{(i)}, y^{(i)}) = \mathcal{L}_{NLL}(g^{(i)}, y^{(i)})$$

hence we wish to minimize the

$$J(\Theta) = \left( \frac{1}{n} \sum_{i=1}^n \mathcal{L}_{NLL}(\sigma(\theta^T x^{(i)} + \theta_0), y^{(i)}) \right) + \lambda \|\theta\|^2$$

note that we didn't include  $\theta_0$  in the regularizer as we don't want to punish large  $\theta_0$ .

## 3.2 Gradient descent

### 3.2.1 Batch gradient descent

---

**Algorithm 5:** batch gradient descent ( $f, \nabla f, \theta_{\text{init}}, \eta, \epsilon$ )

---

$\theta^{(0)} = \theta_{\text{init}}$

$t = 0$

**repeat**

$\theta^{(t)} = \theta^{(t-1)} - \eta \nabla f(\theta^{(t-1)})$

**until**  $|f(\theta^{(t)}) - f(\theta^{(t-1)})| < \epsilon$

**return**  $\theta$

---

**Theorem 3.1.** *If  $f$  is convex, for any desired accuracy  $\epsilon$  there is some  $\eta$  such that batch gradient descent will converge to  $\theta$  within  $\epsilon$  of the optimum.*

Lets derive the closed form of gradient descent for the logistic regression  $J$  function.

$$\nabla J(\Theta) = \left( \frac{1}{n} \sum_{i=1}^n \nabla \mathcal{L}_{NLL}(\sigma(\theta^T x^{(i)} + \theta_0), y^{(i)}) \right) + \lambda \nabla \|\theta\|^2$$

we have that

$$\begin{aligned}\frac{\partial \mathcal{L}_{NLL}(g^{(i)}, y^{(i)})}{\partial \theta_j} &= -\frac{y^{(i)}}{g^{(i)}} \frac{\partial g^i}{\partial \theta_j} + \frac{1 - y^{(i)}}{1 - g^{(i)}} \frac{\partial g^i}{\partial \theta_j} \\ &= \frac{g^{(i)} - y^{(i)}}{g^i (1 - g^{(i)})} \frac{d\sigma(z)}{dz} \frac{\partial z}{\partial \theta_j}\end{aligned}$$

for simplicity assume  $x_0^{(i)} = 1$

$$\begin{aligned}&= \frac{g^{(i)} - y^{(i)}}{g^i (1 - g^{(i)})} \left( g^{(i)} - (g^{(i)})^2 \right) x_j^{(i)} \\ &= (g^{(i)} - y^{(i)}) x_j^{(i)} \\ \implies \nabla \mathcal{L}_{NLL}(g^{(i)}, y^{(i)}) &= (g^{(i)} - y^{(i)}) x^{(i)}\end{aligned}$$

also for the regularizer

$$\frac{\partial \|\theta\|^2}{\partial \theta_j} = \begin{cases} 2\theta_j & j \neq 0 \\ 0 & j = 0 \end{cases}$$

Therefore,

$$\begin{aligned}\nabla_{\theta} J(\Theta) &= \left( \frac{1}{n} \sum_{i=1}^n (g^{(i)} - y^{(i)}) x^{(i)} \right) + 2\lambda \theta \\ \nabla_{\theta_0} J(\Theta) &= \frac{1}{n} \sum_{i=1}^n g^{(i)} - y^{(i)}\end{aligned}$$

### 3.2.2 Stochastic gradient descent

Suppose we want to minimize  $f(\Theta)$  which can be written as a sum of some  $f_i(\Theta)$

$$f(\Theta) = \sum_{i=1}^n f_i(\Theta)$$

Instead of adding up the gradient for each part every iteration, we can choose one of the functions and apply the gradient descent for that function only. We expect that running this algorithm for long enough gives us the optimum solution just like the gradient descent.

---

**Algorithm 6:** stochastic gradient descent  $(f, \nabla f_1, \dots, \nabla f_n, \Theta_{\text{init}}, \eta, T)$

---

```

 $\Theta^{(0)} = \Theta_{\text{init}}$ 
for  $t = 1 \rightarrow T$  do
   $i = \text{Random}(\{1, \dots, n\})$ 
   $\Theta^{(t)} = \Theta^{(t-1)} - \eta(t) \nabla f_i(\theta^{(t-1)})$ 
return  $\theta$ 

```

---

**Theorem 3.2.** *If  $f$  is convex and  $\eta(t)$  is sequence satisfying*

$$\sum_{i=1}^n \eta(t) = \infty \quad \text{and} \quad \sum_{i=1}^n \eta^2(t) < \infty$$

*Then SGD converges almost sure to the optimal  $\Theta$ .*

### 3.3 Linear regression

In regression our output is a real number as oppose to a discrete value. Typically, we use *squared error* for loss function.

$$L(g^{(i)}, y^{(i)}) = (g^{(i)} - y^{(i)})^2$$

and hence our  $J$  function will be the *mean squared error*

$$J(\Theta) = \frac{1}{n} \sum_{i=1}^n (g^{(i)} - y^{(i)})^2 + \lambda R(\Theta)$$

#### 3.3.1 Linear hypothesis

Discarding  $\theta_0$  (for simplicity) and the regularizer term and by allowing our hypothesis  $h$  to be linear — that is  $g^{(i)} = h(x^{(i)}; \theta) = \theta^T x$  —, we can easily arrive at a closed form optimal solution. To do this consider the following definitions

$$X = \begin{bmatrix} x_1^{(1)} & \dots & x_d^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(n)} & \dots & x_d^{(n)} \end{bmatrix} \quad Y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

then we can write  $J(\theta)$  as

$$J(\theta) = \frac{1}{n} (X\theta - Y)^T (X\theta - Y)$$

taking the gradient gives

$$\begin{aligned} \nabla J(\hat{\theta}) &= \frac{2}{n} X^T (X\hat{\theta} - Y) = 0 \\ \implies X^T X \hat{\theta} &= X^T Y \implies \hat{\theta} = (X^T X)^{-1} X^T Y \end{aligned}$$

Assuming the invertibility of  $X^T X$ , this minimizes the  $J$  function. However, we will run into the problem of *overfitting*. By adding the regularizer back we can solve the problem of invertibility and overfitting at the same time. Consider the following  $J_{\text{ridge}}$  function,

$$\begin{aligned} J_{\text{ridge}}(\theta) &= \frac{1}{n} (X\theta - Y)^T (X\theta - Y) + \lambda \theta^T \theta \\ \implies \nabla J_{\text{ridge}}(\hat{\theta}) &= \frac{2}{n} X^T (X\hat{\theta} - Y) + 2\lambda \hat{\theta} = 0 \\ \implies (X^T X + n\lambda I) \hat{\theta} &= X^T Y \implies \hat{\theta} = (X^T X + n\lambda I)^{-1} X^T Y \end{aligned}$$

A faster way of computing the optimal solution is using the SGD or BGD, which since the  $J_{\text{ridge}}$  is convex will converge.

#### 3.3.2 Non-linear hypothesis

A simple generalization to the non-linear case can be achieved by transforming the data using basis functions,  $\phi_i : \mathbb{R}^d \rightarrow \mathbb{R}$

$$h(x; \theta) = \theta_0 + \sum_{i=1}^m \theta_i \phi_i(x)$$

Some examples of basis function include

**Polynomial (univariate)**  $\phi_i(x) = x^i$  for  $i = 1, \dots, m$

$$h(x; \theta) = \sum_{i=0}^m \theta_i x^i$$

**Gaussian** prediction based on similarity to *prototypes*,  $\mu_1, \dots, \mu_m$ , where  $\sigma_i^2$  control how quickly it vanished as a function of the distance to the prototype.

$$\phi_i(x) = \exp\left(-\frac{(x - \mu_i)^2}{2\sigma_i^2}\right)$$

**Sigmoid**

$$\phi_i(x) = \sigma\left(\frac{\|x - \mu_i\|}{\sigma_i}\right)$$

**Fourier basis**

Deriving the optimal solution is like that case of linear hypothesis. Lets define the following matrices

$$\Phi = \begin{bmatrix} 1 & \phi_1(x^{(1)}) & \dots & \phi_m(x^{(1)}) \\ 1 & \phi_1(x^{(2)}) & \dots & \phi_m(x^{(2)}) \\ \vdots & \vdots & \ddots & \vdots \\ 1 & \phi_1(x^{(n)}) & \dots & \phi_m(x^{(n)}) \end{bmatrix} \quad Y = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(n)} \end{bmatrix}$$

Then our objective  $J(\theta)$  can be written as

$$\begin{aligned} J_{\text{ridge}}(\theta) &= \frac{1}{n}(\Phi\theta - Y)^T(\Phi\theta - Y) + \lambda\theta^T\theta \\ \implies \nabla J_{\text{ridge}}(\hat{\theta}) &= \frac{2}{n}\Phi^T(\Phi\hat{\theta} - Y) + 2\lambda\hat{\theta} = 0 \\ \implies (\Phi^T\Phi + n\lambda I)\hat{\theta} &= \Phi^TY \implies \hat{\theta} = (\Phi^T\Phi + n\lambda I)^{-1}\Phi^TY \end{aligned}$$

## 3.4 Regularizers

### 3.4.1 Overfitting

Overfitting occurs when our hypothesis is tuned to the training data but it fails to generalize. Two main causes of overfitting are *model complexity* and low number of training data. Therefore, one way to solve overfitting is to increase training data size. Some other systematic solutions to the overfitting problem include

- Selection of suitable model complexity and good parameters. One way to evaluate a good model is *hold-out* method, where we divide data into training and validation sets then we train a set of models and evaluated their performance on the validation set. However, this method is very wasteful of data and a small validation set obtains relatively noisy estimate of performance. A better method is the previously discussed cross validation.
- Adding a regularizer term. As discussed, a regularizer encourages the parameter to closer to some special value  $\theta_{\text{prior}}$ . When increase  $\lambda$  we tend to increase structural error but decrease estimation error and vice versa.

### 3.4.2 Errors

There are two kinds of way that a hypothesis might contribute to the error on test data.

**Structural error** There is no hypothesis in our class that performs well with the training data.

$$\min \mathbb{E}[(y - w^T x)^2]$$

**Estimation error** The training data was not a good representaion of the test data and hence our hypothesis is not as good.

$$\begin{aligned} \min \mathbb{E}[(w^{*T} x - \hat{w}^T x)^2] \\ w^* = \operatorname{argmin} \mathbb{E}[(y - w^T x)^2] \\ \hat{w} = \operatorname{argmin} \sum_{i=1}^n (y^{(i)} - w^T x^{(i)})^2 \end{aligned}$$

**Expected error** is the sum of the structural error and estimation error.

### 3.4.3 Bias-variance trade-off

need more, chapter 1.1 , 1.3 , 3.1, 3.2 and chapter 2.3, 3.2, 3.4



---

# Chapter 4

## Neural Network

### 4.1 Introduction

A neural network consists of basic units called **neuron** - it is also called a *unit* or a *node*. A node takes an input  $x \in \mathbb{R}^m$  and outputs a single value  $a \in \mathbb{R}$  as such

$$a = f\left(\sum_{i=1}^m x_i w_i + w_0\right)$$

where  $w = (w_1, \dots, w_m)$  are the *weights*,  $w_0$  is the *offset*, and  $f : \mathbb{R} \rightarrow \mathbb{R}$  is called the *activation function* which is not necessarily linear. The pair  $(w, w_0)$  are the parameters that we need to tune to minimize the objective function

$$J(w, w_0) = \sum_{i=1}^n L(NN(x^{(i)}; w, w_0), y^{(i)})$$

We will consider the *feed-forward* networks where the output does not feed back in to input. We will organize the network into *layers* of parallel nodes. Thus, a layer takes an input  $x \in \mathbb{R}^m$  and outputs  $A \in \mathbb{R}^n$  as such

$$A = f(Z) = f(W^T x + W_0)$$

where  $W \in \mathcal{M}_{m \times n}(\mathbb{R})$  is the weight matrix,  $W_0 \in \mathbb{R}^n$  is the offset vector, and is the *pre-activation* vector. Note that, the activation function is applied element-wise. Let's denote the weight matrix, offset vector, pre-activation vector, and output of  $l_{th}$  layers as  $W^l, W_0^l, Z^l, A^l$  and let  $A^0 = X$  then

$$A^l = f(Z^l) = f(W^{lT} A^{l-1} + W_0^l)$$

#### 4.1.1 Activation function

It is obvious that letting  $f$  be linear function makes  $A^1$  a linear function of  $X = A^0$  and hence  $A^L$ , the last layer, will be a linear function of  $X$  which is equivalent to a linear classification or linear regression problem. Some good non-linear functions are

**Step function**

$$\text{step}(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$$

**Rectified linear unit**

$$\text{ReLU}(x) = \max(0, x) = \begin{cases} x & x > 0 \\ 0 & x \leq 0 \end{cases}$$

**Sigmoid**

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

**Hyperbolic tangent**

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1$$

**Softmax function** is a function of  $\mathbb{R}^n \rightarrow \mathbb{R}^n$  with output  $Y \in [0, 1]^n$  that has the property  $\sum Y_i = 1$

$$\text{softmax}(X) = \begin{bmatrix} \frac{\exp(x_1)}{\sum \exp(x_i)} \\ \vdots \\ \frac{\exp(x_n)}{\sum \exp(x_i)} \end{bmatrix}$$

ReLU is common in internal, *hidden*, layers, sigmoid is used in the last layer for binary classification and softmax is used for multiclass classification.

**4.1.2 Error back-propagation**

We wish to use the gradient descent methods to minimize our objective function. To this, we need to compute the  $\nabla_W L(NN(x; W), y)$  where  $W$  represents the all weights  $W^l, W_0^l$  for  $l \in \{1, \dots, L\}$ .

$$\begin{aligned} \frac{\partial L}{\partial W^l} &= A^{l-1} \left( \frac{\partial L}{\partial Z^l} \right)^T \\ \frac{\partial L}{\partial Z^l} &= \frac{\partial L}{\partial Z^{l+1}} \frac{\partial Z^{l+1}}{\partial A^l} \frac{\partial A^l}{\partial Z^l} \\ \frac{\partial L}{\partial Z^L} &= \frac{\partial L}{\partial A^L} \frac{\partial A^L}{\partial Z^L} \end{aligned}$$

$\frac{\partial L}{\partial A^L}$  is  $n^L \times 1$  and depends on particular loss function.

$\frac{\partial Z^l}{\partial A^{l-1}}$  is  $m^l \times n^l$  and is just  $W^l$ .

$\frac{\partial A^l}{\partial Z^l}$  is  $n^l \times n^l$  is equal to  $\left[ \frac{\partial A_i^l}{\partial Z_j^l} \right]$  and therefore it is a diagonal matrix whose elements are equal to  $f'(Z_i^l)$ .

**4.2 Training**

Using the SGD, with initial values that are small enough - since the gradient of activation function tends to zero at larger values- we have

---

**Algorithm 7:** SGD neural network  $(\mathcal{D}_n, T, L, \eta, (m^1, \dots, m^L), (f^1, \dots, f^L))$ 


---

```

for  $l = 1 \rightarrow L$  do
   $W^l \sim \mathcal{N}\left(0, \left(\frac{1}{m^l}\right)^2\right)$ 
   $W_0^l \sim \mathcal{N}(0, 1)$ 
for  $t = 1 \rightarrow T$  do
   $i = \text{Random}(\{1, \dots, n\})$ 
   $A^0 = x^{(i)}$ 
  for  $l = 1 \rightarrow L$  do                                     // forward feed
     $Z^l = W^{lT} A^{l-1} + W_0^l$ 
     $A^l = f^l(Z^l)$ 
   $\text{loss} = L(A^L, y^{(i)})$ 
   $\frac{\partial L}{\partial A^L} = \frac{\partial L}{\partial A^L}(\text{loss})$ 
  for  $l = L \rightarrow 1$  do                                     // error back propagation
    if  $l = L$  then
       $\frac{\partial L}{\partial Z^l} = \frac{\partial L}{\partial A^L} \frac{\partial A^L}{\partial Z^L}$ 
    else
       $\frac{\partial L}{\partial Z^l} = \frac{\partial L}{\partial Z^{l+1}} W^{l+1} \frac{\partial A^l}{\partial Z^l}$ 
       $\frac{\partial L}{\partial W^l} = \frac{\partial L}{\partial Z^l} \frac{\partial Z^L}{\partial W^L}$ 
       $\frac{\partial L}{\partial W_0^l} = \frac{\partial L}{\partial Z^l} \frac{\partial Z^L}{\partial W_0^L}$ 
       $W^l = W^l - \eta(t) \frac{\partial L}{\partial W^l}$ 
       $W_0^l = W_0^l - \eta(t) \frac{\partial L}{\partial W_0^l}$ 

```

---