Supervised Learning

Gabriel Magill*
(Dated: November 12, 2017)

In these set of notes, we introduce concepts relating to machine learning. In particular, we will see techniques relating to classification and regression, with the goal of building up to neural networks. These notes were presented at the McMaster University Theoretical Physics Journal Club.

I. DISCLAIMER

These set of notes are based on the course "CS489/698 - Winter 2017 Introduction to Machine Learning", taught be Pascal Poupart at the University of Waterloo. Material is also drawn from Stuart Russell and Peter Norvig, Artificial Intelligence: A Modern Approach (3rd Edition) (2010), and Christopher Bishop, Pattern Recognition and Machine Learning (2006).

II. INTRODUCTION

The general idea of machine learning is for computers to approximate true functions f by h, where

$$f : X \to Y. \tag{1}$$

In the equation above, X is the input space of features, and Y is the target space of answers. For examples, X can be colours of pixels of a picture, and Y can be what number (1-10) is written on the picture. The determination of h can be done via three methods:

- Supervised Learning: We have a training set of features X with corresponding correct answers Y. We try to infer general rules.
- Reinforcement learning: We have a way of associating penalty terms ϵ given wrong answers, but we don't know a-priori what the answers are.
- Unsupervised learning: finding patterns in X without any answers Y, for example uncovering trends in various facebook age groups.

The target space Y falls into two categories (regardless of the nature of the input space X). The first category is Classification. In this, Y falls into a finite set of classes. Examples of this is image recognition, restaurant recommendations, sunny or rainy temperature. The nature of X is irrelevant. The second category is regression, in which Y is a continuous number. Examples of this are probability estimations, temperature in degrees (actual temperature T). In these notes, we will overview general classification and regression algorithms. We will focus mostly on supervised learning.

* gmagill@perimeterinstitute.ca

III. GENERAL IDEAS

A. Partitioning the Data Set

In supervised learning, we usually have a training set, a test set, and actual new data. The training set is used to develop h (ex: studying the answers of previous midterms), the test set is used to see how well h approximates f (ex: trying to solve a practice midterm), and actual new data (ex: sitting the midterm).
Common problem: Overfitting happens when your accuracy on the training set is very high, but your accuracy on a test set is low. If your h is way too complicated, you start capturing random fluctuations in the training set. Alternatively, if the form of h is too simple, you might underfit.

B. Cross-Validation

Often, one has free parameters λ in $h(\lambda)$ that need to be optimized. *Bad way of doing this:* Developing a hypothesis h based on a training set, testing on the test set, going back to the training set to re-tweak λ, re-testing it on the test set is a classic example of overfitting. The way to avoid this is via cross-validation. In cross-validation, we do:

- Collect all of our supervised data $\{X, Y\}$ into one pool.
- Select (for example) the first 10% of the data to be the test set and the remaining to be training set.
- For a given choice of parameter λ, train $h(\lambda)$ and test on 10%.
- Rince and repeat for remaining 9 partitions of data. Take average accuracy.
- Vary λ and repeat these steps.
- Select λ that gave highest average accuracy.

The above algorithm ensures that we are not optimizing λ erroneously based on any anomaly of a particular partition.

IV. CLASSIFICATION

A. k-nearest neighbour

Let's look now to an example that will allow us to quickly do impressive things very easily (and inefficiently). This is the k-nearest neighbour algorithm. This is defined as:

$$\begin{aligned} h(x) &= y_{x^*} \\ \text{where } x^* &= \text{argmin}_{x'} d(x, x'), \\ d(x, x') &= \left(\sum_j^M c_j |x_j - x'_j|^p \right)^{1/p} \end{aligned} \tag{2}$$

The easiest way to understand this algorithm is by imagining the training set X to be a set of images. Each image entry (rows of X) $\mathbf{x}_i \in X$ is the colour of a pixel. The Y is whether the image is a number from 0-10. In the above, we provide the user with a new image $\mathbf{x}$. They need to figure out what the digit on the picture is (y). They do this by comparing the new picture to each picture $\mathbf{x}_i$ in the training set, and find the picture $\mathbf{x}^*$ that is closest in measure to the new picture. One can also consider finding the k nearest pictures. The precise value of k is determined by cross validation.

V. BUILDING UP THE FORMALISM FROM SCRATCH

A. Linear Regression

The simplest example of regression is linear regression. The goal here is nicely summarized by:

$$\begin{aligned} \mathbf{w}^* &= \text{argmin}_{\mathbf{w}} L_2(\mathbf{w}) \\ \mathbf{w}^* &= \text{argmin}_{\mathbf{w}} \frac{1}{2} \sum_{n=1}^{N} \left(y_n - h(\mathbf{x}_n) \right)^2 \end{aligned} \tag{3}$$

where

$$h(\mathbf{x}_n) = w_0 + \mathbf{x}_n \cdot \mathbf{w}. \tag{4}$$

$y_n \in Y$, N is the number of $\mathbf{x}_n$ we have in X. This equation can be uniquely solved by taking $\vec{\nabla}_{\mathbf{w}} L_2(\mathbf{w}) = \vec{0}$ and solving N equations. Overfitting can be prevented by $L_2(\mathbf{w}) \to L_2(\mathbf{w}) + \frac{\lambda}{2}||\mathbf{w}||^2$. This is called regularization, and it penalizes making $\mathbf{w}$ arbitrary big given small changes in $\mathbf{x}$. Predictions are now made with $h(\mathbf{w}^*)$.

B. Likelihoods, Priors and Bayes' Theorem

Let's generalize this. It can easily be shown that:

$$\begin{aligned} \mathbf{w}^* &= \text{argmin}_{\mathbf{w}} \frac{1}{2} \sum_{n=1}^{N} \left(y_n - h(\mathbf{x}_n) \right)^2 + \frac{\lambda}{2} ||\mathbf{w}||^2 \\ &= \text{argmax}_{\mathbf{w}} k \Pi_n e^{-\frac{\mathbf{w}^T \cdot \Sigma^{-1} \cdot \mathbf{w}}{2}} e^{-\frac{-(y_n - \mathbf{w} \cdot \mathbf{x}_n)^2}{2\sigma^2}} \\ &= \text{argmax}_{\mathbf{w}} \text{Normalization} \times P(\mathbf{w}) \times P(Y|X, \mathbf{w}) \end{aligned} \tag{5}$$

The last equation above is a generalization. We have introduced priors $P(\mathbf{w})$, as well as likelihoods $P(Y|X, \mathbf{w})$. We have also introduced the concept of random fluctuations parameterized by σ as well as independent and identically distributed measurements (i.i.d.) which allows us to factorize the likelihood function. Using Bayes' theorem, we can rewrite this in terms of the A Posteriori function:

$$\mathbf{w}^* = \text{argmax}_{\mathbf{w}} P(\mathbf{w}|X, Y) \tag{6}$$

Assuming we can solve the above equation, we want to make a prediction given new data. The set of $\{\mathbf{w}^*\}$ gives us our hypothesis h. We can use this to make a prediction about an unknown quantity Q given evidence or data $\mathbf{e}$:

$$Pr(Q, \mathbf{e}) = \sum_i Pr(Q|h_i) P(h_i|\mathbf{e}) \tag{7}$$

In general, this is complicated. Typically, we say

$$Pr(Q|\mathbf{e}) \approx Pr(Q|h_{MAP}) \tag{8}$$

where $h_{MAP}(\mathbf{w}^*)$ (Maximum A Posteriori) is the solution to Eq. (6). A variation of this involves making predictions using $Pr(Q|h_{ML})$ (maximum likelihood), that is to say using Bayes' theorem and ignoring the priors. This called method of maximum likelihood. In the large data limit, both methods converge to the same answer, because whatever initial choice of Prior gets washed out.

C. Logistic Regression

I didn't have time to write much about this. The idea is that if you are doing classification, and you have two classes, for a general class of probability distributions, you can prove that the a posteriori function looks like:

$$P(\mathbf{w}|\mathbf{x}) = \frac{1}{1 + \exp(-\mathbf{w} \cdot \mathbf{x})} \equiv \sigma(\mathbf{w} \cdot \mathbf{x}) \tag{9}$$

This is a sigmoid function. If you plot $\sigma(a)$, you see that it interpolates between 0 and 1. So if it's greater than 0.5, you say it's in class A. If it's smaller than 0.5, you say it's in class B. The generalization to many classes is called the soft-max. Furthermore, instead of thinking about the likelihood, you can directly optimize the posteriori

function by taking derivatives on the sigmoid. For the entire data set, the optimization problem at hand is:

$$\mathbf{w}^* = \text{argmax}_{\mathbf{w}} \Pi_{n=1}^{N} \sigma(\mathbf{w}^T \cdot \mathbf{x}_n)^{y_n} (1 - \sigma(\mathbf{w}^T \mathbf{x}_n))^{1-y_n} \tag{10}$$

In the above, $y_A = 1$ and $y_B = 0$. This method is apparently used by all major companies for App recommendations.

VI. GENERALIZED LINEAR FUNCTIONS

In the above, we assumed $h(\mathbf{x}) = \mathbf{w} \cdot \mathbf{x}$. In order words, our hypothesis is linear in the input features $\mathbf{x} \in X$. Is this a good assumption? Not always. What if our true function $f(x) = a + bx + cx^2 + dx^3$? The way to solve this is to use non-linear basis functions. Let $\boldsymbol{\phi}(x) = \{1, x, x^2, x^3\}$. Now, all the above framework is still valid, except that everything is done in terms of:

$$h(\mathbf{x}) = \mathbf{w} \cdot \boldsymbol{\phi}(\mathbf{x}) \tag{11}$$

We have traded non-linear basis for a linear basis, at the expense of increasing the size of the basis. Examples of $\boldsymbol{\phi}(\mathbf{x})$ for $\mathbf{x} \in X$ are $\phi_1 = x_1 * x_1$, $\phi_2 = x_2 * x_1$, $\phi_3 = x_3^4 * x_1^2 + x_5$, etc. The obvious question now is how does one choose the basis ϕ? What size is the new basis? Is it finite or infinite? The power of machine learning is that *one never needs to define the basis* $\boldsymbol{\phi}$. In other words, we never have to specify the non-linear feature space based on X. We will see two ways of how this is realized. The first is Kernel methods. The second is neural networks.

A. Kernel Methods

Let us revisit our error function which we are trying to minimize:

$$L_2(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} (\mathbf{w}^T \cdot \boldsymbol{\phi}(\mathbf{x}_n) - y_n)^2 + \frac{\lambda}{2} \mathbf{w}^T \cdot \mathbf{w} \tag{12}$$

Minimizing this via $\vec{\nabla}_{\mathbf{w}} L_2(\mathbf{w}) = 0$ reveals:

$$\mathbf{w} = -\frac{1}{\lambda} \sum_{n} (\mathbf{w}^T \boldsymbol{\phi}(\mathbf{x}_n) - y_n) \boldsymbol{\phi}(\mathbf{x}_n) \tag{13}$$

In other words, $\mathbf{w}$ is a linear combination of the generalized feature space basis. So instead of considering $\mathbf{w}$, let me replace:

$$\begin{aligned} \mathbf{w} &= \vec{\boldsymbol{\phi}} \cdot \mathbf{a} \\ &\equiv (\boldsymbol{\phi}(\mathbf{x}_1) \cdots \boldsymbol{\phi}(\mathbf{x}_N)) \cdot (a_1 \cdots a_N)^T \end{aligned} \tag{14}$$

Let us define:
$\mathbf{K} = \vec{\boldsymbol{\phi}}^T \vec{\boldsymbol{\phi}}$
$k(\mathbf{x}, \mathbf{x}') = \boldsymbol{\phi}(\mathbf{x})^T \boldsymbol{\phi}(\mathbf{x}')$.

We can now minimize L_2 with respect to $\mathbf{a}$. We find:

$$\mathbf{a} = (\mathbf{K} + \lambda \mathbf{I})^{-1} \mathbf{y} \tag{15}$$

We combine everything together. Our goal is to make a prediction. Hence we have:

$$\begin{aligned} y_* &= \boldsymbol{\phi}(x_*)^T \mathbf{w} \\ &= \boldsymbol{\phi}(\mathbf{x}_*)^T \vec{\boldsymbol{\phi}} \mathbf{a} \\ &= k(\mathbf{x}_*, X)(\mathbf{K} + \lambda I)^{-1} \mathbf{y} \end{aligned} \tag{16}$$

Can we construct $k(\mathbf{x}, \mathbf{x}')$ directly without knowing $\boldsymbol{\phi}$? Yes! Common kernel:

$$k(\mathbf{x}, \mathbf{x}') = \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\sigma^2}\right) \tag{17}$$

Can also use polynomial kernel. With this choice, $\mathbf{K}$ is an $N \times N$ matrix where $K_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$. We can now make predictions! The main trick in all of this is the realization that despite a very large basis, predictions only depend on dot products of basis functions, which is just a number. Hence complexity depends on the amount of data now, and not on the size of the basis.

B. Neural Networks

Neural networks try to answer the same question as we've been seeing all this time. Given a training set X, can I train the correct weights of a function $h(\mathbf{w}, \mathbf{x})$? We started by considering linear regression. We then generalized this to a non-linear basis, and found an alternative way of dealing with the weights in which we didn't have to specify the feature basis.

Neural Networks can be thought of a way of automatically learning what the feature basis is! The various layers of a neural network correspond to various features. We don't have to explicitly specify what these features are. The neural network algorithm takes care of identifying the most defining and useful features by systematically adjusting the weights.

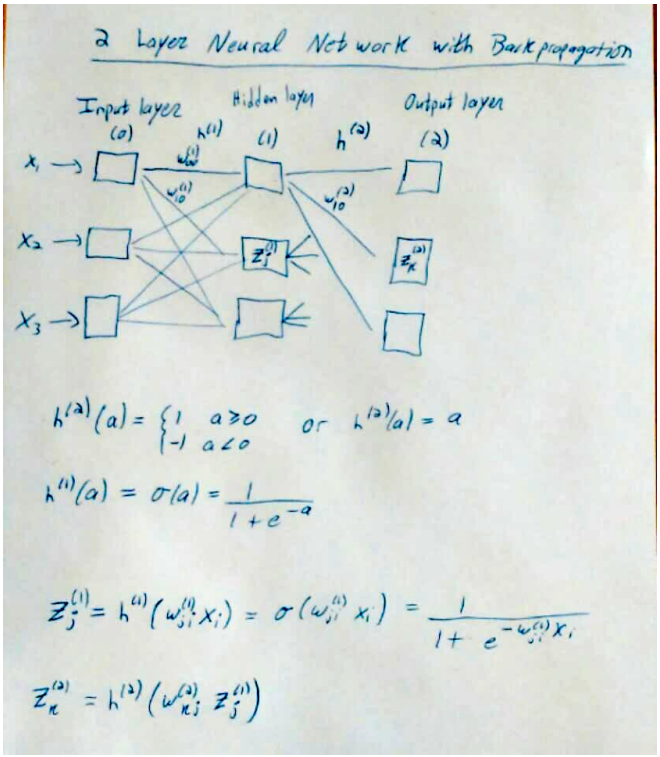
We illustrate this concept with a two-layer neural net. We have input units corresponding to X, hidden units, and output units:

The hidden unit corresponds to a hidden feature (analogous to $\boldsymbol{\phi}$ in the Kernel Method section). The initial units correspond to the given feature space X. The output units maps to a set of specific finite classes . So how do we set the weights? Consider again the error function:

$$L_2(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} ||h(\mathbf{x}_n, \mathbf{w}) - y_n||^2 \tag{18}$$

Now,

$$h(\mathbf{x}, \mathbf{w}) = w_{kj}^{(2)} \sigma(w_{ji}^{(1)} x_i) \tag{19}$$



where we have converted to Einstein summation notation and dropped the bold notation when specifying subscripts. Our goal is always to minimize L_2. We do this as follows:

- Pick arbitrary initial weights $\mathbf{w} = \{\mathbf{w}^{(1)}, \mathbf{w}^{(2)}\}$.
- Consider the training set X. For a given $\mathbf{x}_m \in X$, evaluate $h(\mathbf{x}_m, \mathbf{w})$ for each final class k.
- If its prediction is in agreement with y_m, leaves the weights as is. If it is not in agreement, modify the weights.

How do we confirm if the prediction is in agreement with y_m? If we utilize $h^{(2)}(a) = \{1, -1\}$ if $\{a > 0, a < 0\}$, then predicting an output node of class k with value h of 1 is considered correct and value -1 is considered incorrect when the true result is y_k corresponding to output node k. If we get the wrong answer, back-propagation!

$$\begin{aligned} w_{ji} &\leftarrow w_{ji} - \eta \frac{\partial L_2}{\partial w_{ji}} \\ \frac{\partial L_2}{\partial w_{ji}} &= \frac{\partial L_2}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \equiv \delta_j z_i \end{aligned} \tag{20}$$

Two cases: w_{ji} is an output node ($w_{ji}^{(2)}$), or it's a hidden node ($w_{ji}^{(1)}$).

$$\delta_j = \begin{cases} h'(a_j)(z_j - y_j) & \text{base case: } j \text{ is an output unit} \\ h'(a_j) w_{kj} \delta_k & \text{recursion: } j \text{ is a hidden unit} \end{cases}$$

Since $a_j = w_{ji} z_i$, then $\frac{\partial a_j}{\partial w_{ji}} = z_i$.

C. Deep Learning

It turns out that the above algorithm starts failing when we consider more than 1 hidden layer. The reason for this is because the weights close to the output nodes change fast, but the nodes close to the input notes change extremely slowly, known as the vanishing gradient problem. In fact, one can easily check that given a sigmoid function σ, nesting a bunch of them and considering the above algorithm leads to a very small gradient. The vanishing gradient problem was a major bottleneck up until the past 5 years. The problem was partly solved by considering a different $h^{(i)}$ than the sigmoid σ. Current work on image recognition uses up to 150 different layers, and achieves a better image recognition error rate (3%) than humans (5%).