Machine Learning

830142

August 2020

1 Introduction

We will be using machine learning techniques to train a computer to recognise if there are cats in the images that we provide it. The images will be $64 \times 64 \times 3$ large as they are in the RGB format. We have 209 training images that we will use to train the computer with. Each training image can be represented as a vector of size $12,288$.

$$x^{(i)} = \begin{pmatrix} q_1^{(i)} \\ q_2^{(i)} \\ \vdots \\ q_{12288}^{(i)} \end{pmatrix}. \tag{1}$$

The set of all images can then be represented as a matrix

$$X = \begin{pmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(209)} \\ \vdots & \vdots & & \vdots \end{pmatrix}. \tag{2}$$

We will be using Logistic Regression to train our computer. In a single training image we have $12,288$ parameters that can vary, these are all the pixels that make up the $64 \times 64 \times 3$ image.

Each image is labeled either as 'cat' or 'non-cat'. Logistic regression works by assigning a single number to each image by calculating $y^{(i)} = w^T x^{(i)} + b$ and sending it through an activation function $\sigma(y^{(i)}) = \frac{1}{1+\exp(-y^{(i)})}$, this function will output a number between 0 and 1, given that the weights w and the bias b are "correctly" chosen, we should get 1 whenever we pass through a 'cat' image and we should get a 0 for a 'non-cat' image.

$$w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_{12288} \end{pmatrix}. \tag{3}$$

Of course, this method is not exact so the best we can hope for is that the sigmoid function $\sigma(y^{(i)})$ will give us an answer close to 1 or close to 0 for either a 'cat' or 'non-cat' image, respectively.

When the algorithm starts, we choose the weights and biases at random so the computer will give us random outcomes and will not be helpful at all at determining whether an image does or does not contain a cat in it.

This is where the 'learning' part comes in, we start with a random assumption of weights and the bias and then we pass all the training data (training images) through the sigmoid function and for each image we obtain a number that is the difference from the truth.

So, for example, if we have a 'cat' image this means we should be getting a 1 from the sigmoid function, but let's say we get $\sigma(y^{(i)}) = 0.31$ so the difference from the true result is 0.69, this number is called a *loss*.

Preferably we would like this number to always be 0 for each image but as we have 209 training images we can imagine that we will always have some losses in our algorithm. Having 209 training images, we can calculate 209 losses, one for each image.

Now you can imagine that what we are looking for is that all of these losses to be as small as possible, therefore we invent a quantity called the *cost* (J) which we will define as

$$J = \frac{1}{m} \sum losses \tag{4}$$

where m is the number of training examples, in our case $m = 209$.

Now we get to the minimisation part of the algorithm. Given our weight and bias we have calculated a single number J, and we can imagine that in the first run of the code with random weights and biases, this number will most likely be relatively large. Therefore, it is now time to update the weights and bias so that on the next run J will be smaller. So some weights might need to be lowered, some might need to be turned up, doing this by hand is obviously impossible (or very tedious at the least). So we need a method that tells us how to alter each weight and the bias in order to reduce the cost function. The goal is for each run we want to get a smaller cost function, the number of runs (or iterations) depends on us, normally we stop when the cost function doesn't change much. Now, I understand that I am being quite vague with my terminology but we will get more concrete with definitions later, right now I am just rying to present the big picture, later we will get into the nitty-gritty details.

This procedure of trying to obtain the smallest possible cost function is called minimisation. There are multiple methods to minimise the cost function, the one that we will go with will be minimisation via gradient-descent.

Gradient-descent is a straight forward method of obtaining a minimum of a function. We can imagine that the function J lives in some highly dimensional space and it is very wiggly, if we specify all the weights and the bias we will find ourselves somewhere in that space with a value $J(w, b)$, standing at that point we would surely know where is up and where is down, or how to get to the bottom of the 'valley' of this function, we would just follow the path of steepest descent and slowly make our way to the bottom of that function. This is what gradient descent is about. We start at some position $J(w_0, b_0)$ and we try to make our way to $J(w_{min}, b_{min})$ where w_{min} and b_{min} is the position of the lowest point of the $J(w, b)$ function.

The disadvantage of this method is that if there exist multiple valleys in the J function, gradient descent will only be sensitive to the local valley that it will be placed in, because it doesn't seek the lowest point of the function but rather the lowest local point of the function. This problem can be managed by choosing multiple starting points for the gradient descent and each of them reporting the lowest point that they got to, and then we choose the 'explorer' which has found the lowest laying valley.

To use gradient descent we need to calculate the gradient of the J function with respect to each weight and the bias, so we need to calculate $\frac{\partial J}{\partial w^{(i)}}$ and $\frac{\partial J}{\partial b}$. Using those gradients we can determine which direction we want to move the weight or the bias. For example if we find that $\frac{\partial J}{\partial w^{(i)}}$ is positive, that means we want to

move in the opposite direction such that

$$w_{new}^{(i)} = w_{old}^{(i)} - \frac{\partial J}{\partial w^{(i)}} \tag{5}$$

but to make this equation slightly more reasonable we will determine a learning rate α which will be some small positive number that will enter Eq. (5) as

$$w_{new}^{(i)} = w_{old}^{(i)} - \alpha \frac{\partial J}{\partial w^{(i)}}. \tag{6}$$

This serves as a bottleneck for how fast the weights and bias get updated. For example, we do not want the parameters to overshoot the valley. α determines the step size of the gradient descent. We call that the learning rate, as it determines how fast the computer learns or how many iterations it takes the computer to reach the minimum point of the function, this should not be too quick and not too slow, so a good learning rate is important but the choice is more or less arbitrary, usual learning rates are around 0.1 or 0.01 but it varies based on different problems.

So far, we have a procedure for updating the parameters using gradient descent where we calculate the gradient of the point we are standing at and then depending whether that gradient is positive or negative we move accordingly with a step size determined by the learning rate.

Now after updating each parameter, we recalculate the cost function $J(w, b)$, we should find that now it is smaller than it was before and we repeat the process. After some time we will find that we will reach a plateau and the cost function will no longer be changing, we have reached a minimum, this can happen after $1,000$ or $10,000$ iterations, but we will surely get there.

Having reached the minimum we now have our weight and bias set as best as we can given our training data. We should save the values of those weights and bias and use them to test how well we have managed to train our computer to recognise images containing cats in them. We could now pass an image the computer has never seen before through the $y = w^T x + b$ function and then through the sigmoid function, and we would obtain a number, let's say there was a cat in that picture and our computer outputs a number of $\sigma(y) = 0.85$ we could treat that as the computer is pretty confident that there is a cat in this image and we would round this number to 1, saying there is a cat in this picture, of course the computer can be wrong but this is how we determine what the computer thinks, values above 0.5 are treated as though the computer believes there is a cat and numbers below mean the computer thinks there is no cat in the image.

This was an introduction into the problem and hopefully I have not used any concepts that are foreign. I have been quite vague in this introduction with certain aspects in order to not confuse the reasoning by adding unnecessary complications. This introduction is a very raw and basic idea of how Logistic Regression can be used to solve a simple machine learning problem. We will now go into more detail in order to prepare one for coding their own machine learning program to distinguish cat images from non-cat images.t

2 The Loss Function

The loss function which we have mentioned previously was a very simplistic function described as the difference between the true value and the one given by the sigmoid function.

$$\mathscr{L}_{\text{simple}} = |y_{true}^{(i)} - \sigma(y^{(i)})|. \tag{7}$$

The above is a simplistic function and not the one commonly used.

The one that we will use will be slightly more complicated but the idea remains, it will still serve as a test for the difference between true value and a predicted one.

$$\mathscr{L}^{(i)} = -y^{(i)}_{true} \log(a^{(i)}) - (1 - y^{(i)}_{true}) \log(1 - a^{(i)}), \tag{8}$$

where $a^{(i)} \equiv \sigma(y^{(i)})$.

Now the cost function reads,

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathscr{L}^{(i)}, \tag{9}$$

We need to be able to calculate the derivative of the cost function for the purposes of updating our parameters. It turns out the solution is quite simple,

$$\frac{\partial J}{\partial w} = \frac{1}{m} X (A - Y)^T, \tag{10}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m} \sum_{i=1}^{m} (a^{(i)} - y^{(i)}_{\mathrm{t}}), \tag{11}$$

where $A = (a^{(1)}, a^{(2)}, ..., a^{(m)})$ and $Y = (y^{(1)}_{\mathrm{t}}, y^{(2)}_{\mathrm{t}}, ..., y^{(m)}_{\mathrm{t}})$, with 't' standing for true.