LOSS FUNCTION OR COST FUNCTION

Machines learn by means of a **loss function**. It's a method of evaluating how well specific algorithm models the given data. If predictions deviates too much from actual results, **loss function** would output a very large number.

For Regression, common loss functions are:

Mean Absolute Error

Mean absolute error, is measured as the average of sum of absolute differences between predictions and actual observations.

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n}$$

Mean Square Error

Mean square error is measured as the average of squared difference between predictions and actual observations.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}$$

For Classification, common loss functions are:

Credits: https://stackoverflow.com/questions/41990250/what-is-cross-entropy

Cross-entropy is commonly used to quantify the difference between two probability distributions. Usually the "true" distribution (the one that your machine learning algorithm is trying to match) is expressed in terms of a one-hot distribution.

For example, suppose for a specific training instance, the label is B (out of the possible labels A, B, and C). The one-hot distribution for this training instance is therefore:

Pr(Class A)	Pr(Class B)	Pr(Class C)
0.0	1.0	0.0

You can interpret the above "true" distribution to mean that the training instance has 0% probability of being class A, 100% probability of being class B, and 0% probability of being class C.

Now, suppose your machine learning algorithm predicts the following probability distribution:

Pr(Class A)	Pr(Class B)	Pr(Class C)
0.228	0.619	0.153

How close is the predicted distribution to the true distribution? That is what the cross-entropy loss determines. Use this formula:

$$H(p,q) = -\sum_{x} p(x)\ \log q(x).$$

Where p(x) is the wanted probability, and q(x) the actual probability. The sum is over the three classes A, B, and C. In this case the loss is **0.479** :

H = - (0.0*ln(0.228) + 1.0*ln(0.619) + 0.0*ln(0.153)) = 0.479

So that is how "wrong" or "far away" your prediction is from the true distribution.

Note that when we are dealing with multiple classes (say, for example predicting the digit from the hand-written images of digits will be a 9 class i.e. 0-9 classification problem) the loss function is commonly referred to as Softmax.

The **cross-entropy** compares the model's prediction with the label which **is** the true probability distribution. The **cross-entropy** goes down as the prediction gets more and more accurate. It becomes zero if the prediction **is** perfect. As such, the **cross-entropy can** be a loss function to train a classification model.

Please remember that Entropy in plain English means the uncertainty associated with a sequence of events. Less entropy signifies more predictable behavior.

Just think as to where the entropy would be more (1) flipping a coin or (2) rolling a die.

Side-note: So, all these Machine Learning problems that we would be solving boils down to optimizing these cost functions that we discussed above. Please note that the cost function should be differentiable so that you can apply techniques like Gradient descent to reach optima; in another words it is about minimizing the error. We have discussed Gradient Descent in our other handout

GRADIENT DESCENT

Please note that this document serves as a reference document to understand Gradient Descent more intuitively. If you need to understand the calculus involved, then here is a great resource: https://ml-cheatsheet.readthedocs.io/en/latest/calculus.html#id34

References:

1. A beautiful explanation of Gradient Descent from the authority - Dan Jurafsky (Professor of Computer Science, Stanford University) from his book Speech & Language Processing - https://web.stanford.edu/~jurafsky/

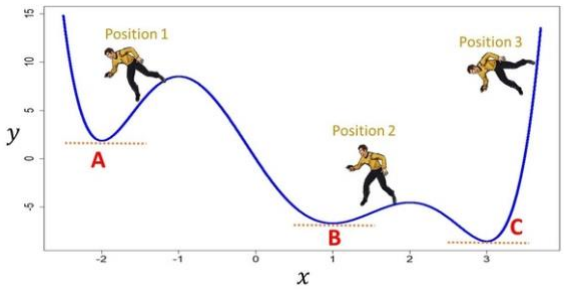
2. Great answer by Saurabh Awasthi on Gradient Descent - https://www.quora.com/What-is-meant-by-gradient-descent-in-laymen-terms

So, what does Gradient mean?

One classic way of finding optimums is by taking the derivative of the objective/loss function, setting it to zero and solve it for your model parameter. If you have more than one parameter, you would have a gradient instead of derivative and your equation becomes a vector equation. The gradient can be thought of as the vector at every point pointing to the next local minimum of your function.

To read more about Gradients and fundamentals of Calculus, here is a great resource:

Imagine you are on the top of the hill and you want to reach the lowest point of the hill. The hill shape is like the below image :



Now assume you are at position 1 and weather is so foggy that you can see only 2 steps at your left and right side. So you see your right and left side and decide to move 2 steps in your left direction as it takes you down to the hill. After moving to your new position you do the same again and move further down to the hill. You end up at point A because from there you can not go down any further. So you decide this is the lowest of the hill.

Now assume you start at position 2. Using the same method you will end up at point B. From B you can not go down further as the area at left and right of the point B is at higher altitude than point B.

Now if you would have started at position 3, then you would have ended up at point C. From point C you can not move down further so you assume this is a lowest point on the hill which actually is.

So depending on from which position you start, you can end up at different lowest points on the hill as you make your decision to move based on the 2 steps at left and right at any position.

In above example, Point A and B are called **Local Minima** and Point C is called **Global Minima**. There are methods to avoid Local Minima like Random Start Gradient Descent.

Gradient Descent is useful in optimization problems as sometimes you are not looking for exact value but for the value which is acceptable in your domain.

Now just consider your loss function; say MSE for Regression and Cross Entropy (that we discussed) for Classification so the ultimate objective would be to use Gradient descent to calculate the optima, in other over simplified words the point where error is minimum.

Learning rate

The size of these steps (for example the steps in the hill example above) is called the *learning rate*. With a high learning rate we can cover more ground each step, but we risk overshooting the lowest point since the slope of the hill is constantly changing. With a very low learning rate, we can confidently move in the direction of the negative gradient since we are recalculating it so frequently. A low learning rate is more precise, but calculating the gradient is time-consuming, so it will take us a very long time to get to the bottom.

Although the algorithm (and the concept of gradient) are designed for direction *vectors*, let's first consider a visualization of the case where the parameter of our system is just a single scalar w, shown in Fig. 5.3.

Given a random initialization of w at some value w^1, and assuming the loss function L happened to have the shape in Fig. 5.3, we need the algorithm to tell us whether at the next iteration we should move left (making w^2 smaller than w^1) or right (making w^2 bigger than w^1) to reach the minimum.

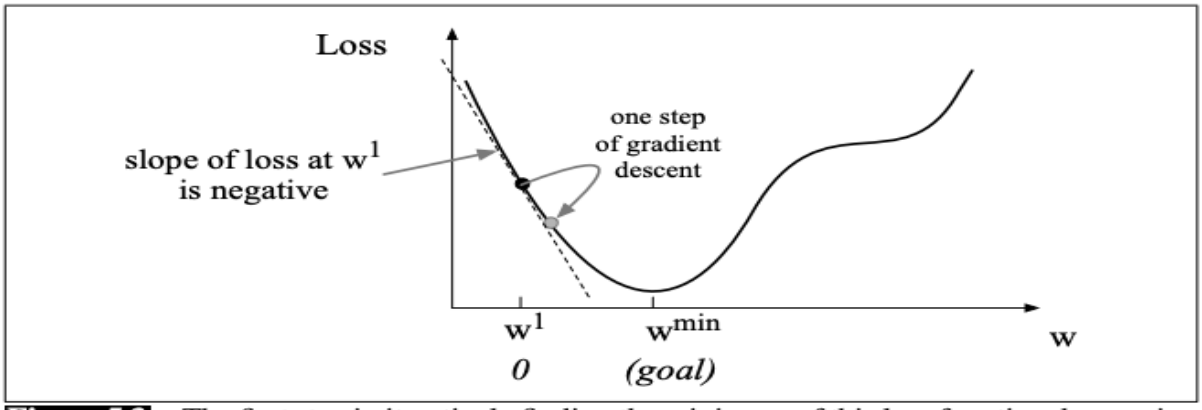


Figure 5.3 The first step in iteratively finding the minimum of this loss function, by moving w in the reverse direction from the slope of the function. Since the slope is negative, we need to move w in a positive direction, to the right. Here superscripts are used for learning steps, so w^1 means the initial value of w (which is 0), w^2 at the second step, and so on.

The gradient descent algorithm answers this question by finding the **gradient** of the loss function at the current point and moving in the opposite direction. The gradient of a function of many variables is a vector pointing in the direction of the greatest increase in a function. The gradient is a multi-variable generalization of the slope, so for a function of one variable like the one in Fig. 5.3, we can informally think of the gradient as the slope. The dotted line in Fig. 5.3 shows the slope of this hypothetical loss function at point $w = w^1$. You can see that the slope of this dotted line is negative. Thus to find the minimum, gradient descent tells us to go in the opposite direction: moving w in a positive direction.

The magnitude of the amount to move in gradient descent is the value of the slope $\frac{d}{dw}f(x;w)$ weighted by a **learning rate** η. A higher (faster) learning rate means that we should move w more on each step. The change we make in our parameter is the learning rate times the gradient (or the slope, in our single-variable example):

$$w^{t+1} = w^t - \eta \frac{d}{dw} f(x;w) \tag{5.13}$$

Now let's extend the intuition from a function of one scalar variable w to many variables, because we don't just want to move left or right, we want to know where in the N-dimensional space (of the N parameters that make up θ) we should move. The **gradient** is just such a vector; it expresses the directional components of the sharpest slope along each of those N dimensions. If we're just imagining two weight dimensions (say for one weight w and one bias b), the gradient might be a vector with two orthogonal components, each of which tells us how much the ground slopes in the w dimension and in the b dimension. Fig. 5.4 shows a visualization:

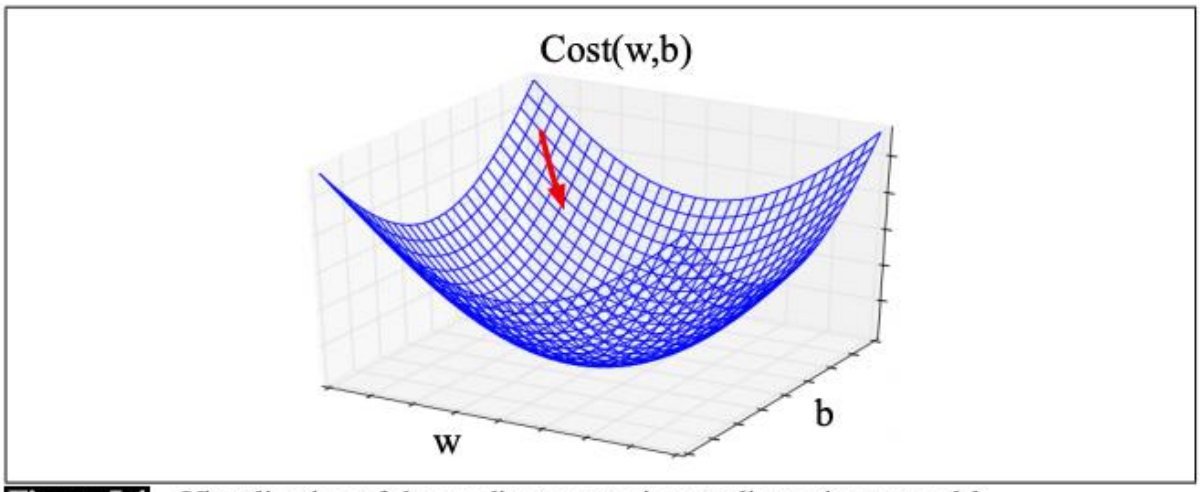


Figure 5.4 Visualization of the gradient vector in two dimensions w and b.

In an actual logistic regression, the parameter vector w is much longer than 1 or 2, since the input feature vector x can be quite long, and we need a weight w_i for each x_i. For each dimension/variable w_i in w (plus the bias b), the gradient will have a component that tells us the slope with respect to that variable. Essentially we're asking: "How much would a small change in that variable w_i influence the total loss function L?"

The learning rate η is a parameter that must be adjusted. If it's too high, the learner will take steps that are too large, overshooting the minimum of the loss function. If it's too low, the learner will take steps that are too small, and take too long to get to the minimum. It is common to begin the learning rate at a higher value, and then slowly decrease it, so that it is a function of the iteration k of training; you will sometimes see the notation η_k to mean the value of the learning rate at iteration k.

References:

Cross-Entropy: https://stackoverflow.com/questions/41990250/what-is-cross-entropy

A beautiful explanation of Gradient Descent from the authority - Dan Jurafsky (Professor of Computer Science, Stanford University) from his book Speech & Language Processing - https://web.stanford.edu/~jurafsky/

Great answer by Saurabh Awasthi on Gradient Descent - https://www.quora.com/What-is-meant-by-gradient-descent-in-laymen-terms