1. Logistic regression gives an output as a probability P(X) where X is the input(s). If we are modeling the probability that an input (X) belongs to the default class (Y=1), we can write this formally as:

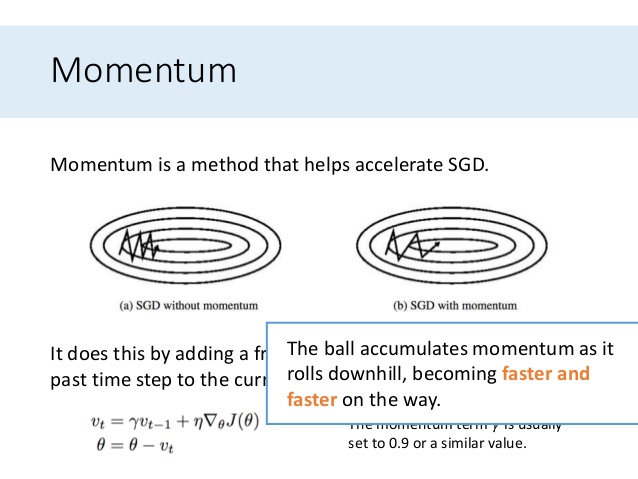
P(X) = P(Y=1|X)

The advantage of using this system, with say an output of 0.7, over binary outputs of 0 and 1 is that this system gives us the probability/accuracy of its prediction, and if we wish we can always convert it to the binary output, by applying a function such 1 if y> 0.5, and 0 otherwise. This gives us a lot of information about our output, rather than just a binary classification

1. SGD with momentum is based on the notion of a new sequence which is based on the average of the previous terms, something like

Vt = bVt-1 + (1 - b)St where S is the original sequence and b belongs to [0,1]

you can think of *beta* as follows. We’re approximately averaging over last *1 / (1- beta)* points of sequence. The intuition is that near a local optima there often exist ravines, where the slope in one direction is much more than in the other direction. What this means is that our learning rate is limited, as we do not want to overshoot or diverge. Taking the exponentially weighted average of the previous terms means that, like in the case of the contour diagram shown, the average of our y oscillations almost add up to zero but along the x they don’t, and our function speeds up in the direction we want it to



3a. The key advantage of using minibatch as opposed to the full dataset goes back to the fundamental idea of stochastic gradient descent[1](https://arxiv.org/abs/1804.07612).

In batch gradient descent, you compute the gradient over the entire dataset, averaging over potentially a vast amount of information. It takes lots of memory to do that. But the real handicap is the batch gradient trajectory land you in a bad spot (saddle point).

In pure SGD, on the other hand, you update your parameters by adding (minus sign) the gradient computed on a *single* instance of the dataset. Since it's based on one random data point, it's very noisy and may go off in a direction far from the batch gradient. However, the noisiness is exactly what you want in non-convex optimization, because it helps you escape from saddle points or local minima. The disadvantage is it's terribly inefficient and you need to loop over the entire dataset many times to find a good solution.

The minibatch methodology is a compromise that injects enough noise to each gradient update, while achieving a relative speedy convergence.

3b. We initialise the weights asymmetrically to solve the problem of symmetry breaking. If each weight in the first layer was the same, every element in the first hidden layer would be the same, as they would undergo the same linear transformation and activation. Due to this, every element in the second hidden layer would also be the same and so on….

This would lead to a lot of redundancy as the same calculation is being done, and there is no point of having multiple neurons in each hidden layer.

3c. What regularization does is add a term l\*(W)^2 to the cost function, so effectively to minimise the cost, we must make W small. Overfitting occurs when we have too many input features, so by making the corresponding weights smaller, we help in reducing the contribution of those features.

4. The formula for CNN is (n + 2p - f)/s + 1

Here p = 1 and f = 3 for both rows and columns, and s = 2

Since the output channels is 8, the image(s) will be of size HxWx8

Where H = (N1 + 2 - 3)/2 + 1 = (N1 + 1)/2

And W = (N2 + 1)/2