Question 1:

Formulation of linear regression as maximization of a likelihood function:

The main mechanism for finding parameters of statistical models like linear regression is known as maximum likelihood estimation(MLE). Our basic goal is that if the given data has been generated by a model, then what parameters were most likely to have been used so that the occurrence(or likelihood) of given data around the hypothesis with inclusion of those parameters is maximized. So, in linear regression we are seeking those values of theta where conditional probability P(data | theta)is maximum. This is known as likelihood of data and we seek to maximize it. For computational ease, we try to find maximum of log-likelihood instead rather than likelihood itself. Also, for further increase in computational ease, it is often easier to minimize the negative of the log-likelihood rather than maximize the log-likelihood itself. Hence, we stick a minus sign in front of the log-likelihood to give us the negative log-likelihood (NLL). So, the function we need to minimize becomes:

$$
\begin{aligned}
\text{NLL}(\theta) &= -\sum_{i=1}^{N} \log p(y_i \mid \mathbf{x}_i, \theta) \\
&= -\sum_{i=1}^{N} \log\left[\left(\frac{1}{2\pi\sigma^2}\right)^{\frac{1}{2}} \exp\left(-\frac{1}{2\sigma^2}(y_i - \beta^T\mathbf{x}_i)^2\right)\right] \\
&= -\sum_{i=1}^{N} \frac{1}{2}\log\left(\frac{1}{2\pi\sigma^2}\right) - \frac{1}{2\sigma^2}(y_i - \beta^T\mathbf{x}_i)^2 \\
&= -\frac{N}{2}\log\left(\frac{1}{2\pi\sigma^2}\right) - \frac{1}{2\sigma^2}\sum_{i=1}^{N}(y_i - \beta^T\mathbf{x}_i)^2 \\
&= -\frac{N}{2}\log\left(\frac{1}{2\pi\sigma^2}\right) - \frac{1}{2\sigma^2}\text{RSS}(\beta)
\end{aligned}
$$

Where RSS is the residual sum of squares, also known as sum of squared error(SSE).
Since the first term in the equation is a constant we simply need to concern ourselves with minimizing the RSS, which will be sufficient for producing the optimal parameter estimate. So if we go on differentiating this RSS w.r.t beta and equating it with zero, we end with optimal parameters to be given in a matrix form as:

$$\hat{\beta}_{\text{OLS}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

This matrix gives the same parameters that we would have got through gradient descent or any other optimization techniques. So, this is how we can formulate linear regression as maximum likelihood estimation.

Why classification algorithms learn a prob.distribution over classes??

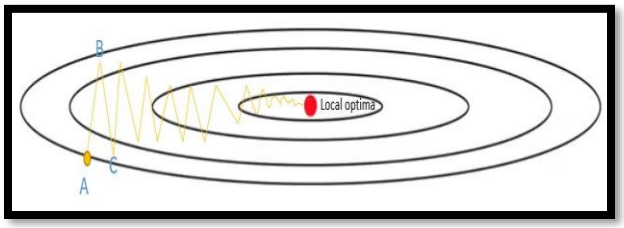
In machine learning, all classification algorithms generally uses a probability distribution as output over several classes instead of classifying them as what binary(0/1) does. This is because in machine learning classification our main task is to predict that to which class the given data will belong to on the basis of model we have developed on past data examples. Now, in this world nothing can be completely deterministic as of what factors and in what amount will influence a data to make it belong to a particular class. If we develop our model seeing on some of the features, then there is a possibility that in future some more features will come which can play as a deciding role in classifying a data to a particular class. So, because of these non-deterministic outcomes expected from a classifier, it is always better to make our classifier probabilistic, i.e. it should give probability of its belonging to a particular class for all classes. In this way, we can find out the class to which it's most likely to belong by seeing the class of maximum probability. So, we can see that a probabilistic model instead of a deterministic one has more chances of predicting the model accurately.

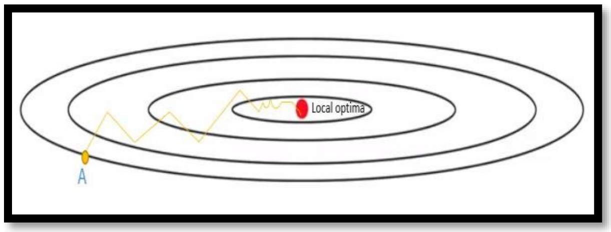
Question 2:

Mini-batch gradient descent makes a parameter update with just a subset of examples, the direction of the update has some variance, and so the path taken by mini-batch gradient descent will "oscillate" toward convergence. However, gradient descent with Momentum considers the past gradients to smooth out the update. It computes an exponentially weighted average of the gradients, and then uses that gradient to update the weights instead. It works faster than the standard gradient descent algorithm.

Example:

Now, consider an example where we are trying to optimize a cost function which has contours like below and the red dot denotes the position of the local optima (minimum).





We start gradient descent from point 'A' and after one iteration of gradient descent we may end up at point 'B', the other side of the ellipse. Then another step of gradient descent may end up at point 'C'. With each iteration of gradient descent, we move towards the local optima with up and down oscillations. If we use larger learning rate then the vertical oscillation will have higher magnitude. So, this vertical oscillation slows down our gradient descent and prevents us from using a much larger learning rate. By using the exponentially weighted average values of dW and db, we tend to average out the oscillations in the vertical direction closer to zero as they are in both directions (positive and negative). Whereas, on the horizontal direction, all the derivatives are pointing to the right of the horizontal direction, so the average in the horizontal direction will still be pretty big. It allows our algorithm to take more straight forwards path towards local optima and damp out vertical oscillations. Due to this reason, the algorithm will end up at local optima with a few iterations.

Implementation:

During backprop, we use dW and db to update our parameters W and b as follows:

W = W-learning rate * dW

b = b-learning rate * db

In momentum, instead of using dW and db independently for each epoch, we take the exponentially weighted averages of dW and db.

$$Vdw = \beta \times Vdw + (1 - \beta) \times dW$$
$$Vdb = \beta \times Vdb + (1 - \beta) \times db$$

where beta ‘β’ is another hyperparameter called momentum and ranges from 0 to 1. It sets the weight between the average of previous values and the current value to calculate the new average. After calculating exponentially weighted averages, we will update our parameters.

$$W = W\text{-learning rate}*Vdw$$
$$b = b\text{-learning rate} * Vdb$$

Intuition:

Imagine a car. The car is going through a mountain range. Being a mountain range, naturally the terrain is hilly. Up and down, up and down. But we, the driver of that car, only want to see the deepest valley of the mountain. So, we want to stop at the part of the road that has the lowest elevation. Only, there’s a problem: the car is just a box with wheels! So, we can’t accelerate and brake at our will, we’re at the mercy of the nature! So, we decided to start from the very top of the mountain road and pray that Newton blesses our journey. We’re moving now! As our “car” moving downhill, it’s gaining more and more speed. We find that we’re going to get through a small hill. Will this hill stop us? Not quite! Because we have been gaining a lot of momentum! So, we pass that small hill. And another small hill after that. And another. And another… Finally, after seems like forever, we find ourselves facing very tall hill. Maybe it’s tall because it’s at the bottom of the mountain range? Nevertheless, the hill is just too much for our “car”. Finally it stops. And it’s true! We could already see the beautiful deepest valley of the mountain! That’s exactly how momentum plays part in SGD. It uses physical law of motion to go pass through local optima (small hills). Intuitively, adding momentum will also make the convergence faster, as we’re accumulating speed, so our Gradient Descent step could be larger, compared to SGD’s constant step.

Question 3:

a) **Why minibatches are used??**

The key advantage of using minibatch as opposed to the full dataset goes back to the fundamental idea of stochastic gradient descent. In batch gradient descent, we compute the gradient over the entire dataset, averaging over potentially a vast amount of information. It takes lots of memory to do that. And the worst happens when batch gradient trajectory land us in a bad spot (saddle point). In pure SGD, on the other hand, we update our 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 we want in non-convex optimization, because it helps us escape from saddle points or local minima. The disadvantage is it's terribly inefficient and we need to loop over the entire dataset many times to find a good solution.

Efficiency and robustness of min-batch:

The minibatch methodology is however a compromise that injects enough noise to each gradient update, while achieving a relative speedy convergence. Typically, especially early in training, the parameter-gradients for different subsets of the data will tend to point in the same direction. So gradients evaluated on 1/100th of the data will point roughly in the same general direction as on the full dataset, but only require 1/100 the computation. Since convergence on a highly-nonlinear deep network typically requires thousands or millions of iterations no matter how good the gradients are, it makes sense to do many updates based on cheap estimates of the gradient rather than few updates based on good ones.

b) If we initialize all the weights to be zero, then all the neurons of all the layers performs the same calculation, giving the same output and thereby making the whole deep net useless. If the weights are zero, complexity of the whole deep net would be the same as that of a single neuron and the predictions would be nothing better than random and our network will not be able to learn something interesting that we sought to do. Nodes that are side-by-side in a hidden layer connected to the same inputs must have different weights for the learning algorithm to update the weights. By making weights as non zero, the algorithm will learn the weights in next iterations and won't be stuck. In this way, breaking the symmetry happens.

c) Regularization relies on the assumption that a model with small weights is simpler than a model with large weights. Thus, by penalizing the square values of the weights in the cost function we drive all the weights to smaller values. It becomes too costly for the cost to have large weights! This leads to a smoother model in which the output changes more slowly as the input changes and thus, our model becomes less prone to overfitting. Some other ways to tackle the problem of overfitting is that we can increase our training data size with the help of **Data Augmentation**. What we do in data augmentation is that we create distortions in some of our data examples by randomly resizing them, cropping the images, or rotating them to a certain angle. This results in increasing our training size and make our model fit to more generalizations of our previous data which decreases the variance surely and thereby prevent overfitting. Another thing we can do is to check the performance of our model on cross-validation sets and stopping the process when the model gives the minimum cross-validation error. This is called **Early Stopping** and this helps in preventing overfitting by selecting a model with parameters which give least variance.

d) We generally normalize the input layer by adjusting and scaling the activations. For example, when we have some features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get 10 times or more improvement in the training speed. This is the idea behind batch normalization. In a neural network, batch normalization is achieved through a normalization step that fixes the means and variances of each layer's inputs. Batch normalization reduces the amount by what the hidden unit values shift around (covariance shift). Also, batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers. We can use higher learning rates because batch normalization makes sure that there’s no activation that’s gone really high or really low. And by that, things that previously couldn’t get to train, it will start to train. It reduces overfitting because it has a slight regularization effects. Similar to dropout, it adds some noise to each hidden layer's activations. Therefore, if we use batch normalization, we will use less dropout, which is a good thing because we are not going to lose a lot of information. Because of all these advantages the batch normalization offers, it stabilizes the training of neural network.

Question 4:

Here, input channel has 3 feature maps and output channel has 8. So, our filter size would not be 3*3 but actually it will be of 3*3*3 and for each output feature map, we'll require a filter of same dimensions. So, total parameters(excluding bias terms) in this layer become **3*3*3*8=216.**

Now, padding(p)=1, stride(s)=2, input size=N1*N2*3, filtersize(f)=3*3*3
So, the output size is: (floor((N1+2*p-f)/s+1))* (floor((N2+2*p-f)/s+1))*8
=floor((N1+1)/2)* floor((N2+1)/2)*(8)