1

Linear Regression as Maximum Likelihood

In linear regression, we have

$$h_\theta(x) = X * theta$$
$$J(\theta) = \frac{1}{2m}(\sum_{i=1}^{m}(h_\theta(x^i) - y^i)^2)$$

Maximum Likelihood Estimation is a probabilistic framework, which estimates the parameters of a model given the data. We usually use the Gaussian distribution model.

In linear regression, we wish to find the line which best fits the data. We can think of this as if x and y have a linear relationship, but with some error included. This error can be modelled as a Gaussian distribution, with mean 0 and variance σ^2

$$y = \theta_1 x + \theta_0 + \epsilon$$
$$\epsilon \sim \mathcal{N}(0, \sigma^2)$$

This model has 3 parameters; θ_1, θ_2, and σ^2. Our goal is to find the best parameters for θ_1 and θ_2. First let's rewrite the model as a single conditional distribution given x:

$$y \sim N(\theta_1 x + \theta_2, \sigma^2)$$

Now we can write the conditional distribution in terms of this Gaussian. Therefore the probability distribution function:

$$f(y|x; \theta_1, \theta_2, \sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\frac{-(y-(\theta_1 x+\theta_0))^2}{2\sigma^2}}$$

Since each point is independent and identically distributed, we can write the likelihood function as the product of individual probability densities. Hence:

$$L_X(\theta_1, \theta_2, \sigma^2) = \frac{1}{(2\pi\sigma^2)^{m/2}} \prod_{(x,y)\in X} e^{\frac{-(y-(\theta_1 x+\theta_0))^2}{2\sigma^2}}$$

Now we have to find the parameters which will maximise this function. To simplify, we take the log of this likelihood, to get rid of most exponents. Also, after taking the log, we see that all terms are negative. THerefore, we take the (-) out, and try to minimise the **negative log likelihood**

$$NLL = -l_X(\theta_1, \theta_2, \sigma^2) = \log(\sqrt{2\pi\sigma^2}) + \frac{1}{2\sigma^2} \sum (y - \hat{y})^2$$

$$\text{Where, } \hat{y} = \theta_1 x + \theta_0$$

Removing the constant terms, we see that the the term that we are trying to minimise is:

$$\sum (y - \hat{y})^2$$

Hence, the maximum likelihood estimate for our linear model is the line which minimises the sum of squared errors. Therefore we can also use maximum likelihood in order to build our model.

Logistic Function

Logistic regression learns a probability distribution instead of directly producing an output y, because:

- Not using any activation function for continous output means that the model can be sensitive to slight changes in the data or the parameters. A very slight change could change the output from 0

to 1, and similiarly give a large change in the cost function, while using a probabilistic output would give a better idea of the real situation.
- Similiarly using a binary output for your loss function would only allow you a few discrete values, and hence there is no true gradient. Hence we will have to use other optimisation algorithms
- In the event of multiclass distribution, it is possible to get many elements as1, and many as 0. Therefore to ensure that we only get one class, we will have to add another parameter b, such that the activation function returns 1 iff x > b. Simpler is to use a probabilistic method, as we can simply choose the maximum, and also gain more information about the way our NN works and how we can better it.

2

Exponentially weighed average

Exponentially weighed avergaes deal with sequence of numbers. Say we have a noisy sequences. We wish to find some sort of a moving average, which would "denoise" the data and bring it closer to the original function.
Exponentially weighted averages define a new sequence V with the following equation:

$$V_t = \beta V_{t-1} + (1-\beta)S_t \qquad \beta \in [0,1]$$

Here β is a hyperparameter which ranges from 0 to 1. Higher value of beta makes the curve smoother, but shifts it to the right, as we average over a large number of examples. On the other hand, smaller beta creates a rough curve which fluctuates a lot. β = 0.90 is considered a good balance.
From the definition above, we can deduce that the value of V_t is dependent on all the previous numbers in the original sequence S. all values of S are assigned some weight. Older values get much smaller weights and therefore contribute very less for the overall value

SGD with Momentum

We define a momentum, which is a moving average of our gradients. We then use it to update the weights of the network.

$$\text{for i} = 1 : \text{iter\_count}$$
$$V_t = \beta V_{t-1} + (1-\beta)\nabla_w L(W, X, y)$$
$$W = W - \alpha V_t$$

Here W_t is a moving average of the gradients, i.e. it is the average of some number of the previous derivatives (usually 10). We directly use the moving average of the derivatives to update our weights and biases.

We can compare this with a physical analogy: Imagine a ball rolling downhill a bowl surface. We can then say that $\nabla_w L(W, X, y)$ is the acceleration of the ball. In normal gradient descent, we find the direction of the acceleration, move slightly in that direction, and stop. However, in gradient-descent with momentum, the previous acceleration is also taken into account, which acts as a sort of momentum, and makes the ball move more smoothly towards the minima. β acts as friction, determining the relative importance of acceleration and momentum.

Tha advantages of using momentum above normal SGD are:

- With Stochastic Gradient Descent we don't compute the exact derivate of our loss function. Instead, we're estimating it on a small batch. So we're not always going in the optimal direction, because our derivatives are 'noisy'. Exponentially weighed averages can provide us a better estimate which is closer to the actual derivate than our noisy calculations.
- Ravine is an area, where the surface curves much more steeply in one dimension than in another. Ravines are common near local minimas in deep learning and SGD has troubles navigating them. SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. Momentum helps accelerate gradients in the right direction.

3

3a

Mini-batches are a method by which we use a small subset of b datapoints in order to calculate the error term for gradient descent. Mini-batch gradient descent is a compromise between batch and stochastic gradient descent.

In batch gradient descent, we calculate the error term for all the training examples and then add all of them for one gradient descent step. Therefore batch gradient descent takes a huge amount of time to train even for a single epoch. However an advantage is that we move smoothly and directly towards the minimum, and it is possible to reach the minimum.

On the other hand, stochastic gradient descent takes less time compared to batch gradient descent, as each descent step requires merely one datapoint. However, it is fluctuautes much more, and

doesn't settle at the minimum. It instead oscillates in a small region around the minimum. Another demerit is that as only one term is used in each step, we cannot use any form of vectorisation to speed up the procedure.

Mini-batch gradient allows us to achieve the advantages of both the former methods.
Instead of either one or all, mini-batch takes b datapoints and uses a vectorised implementation to calculate the mean gradient, and uses this for a descent step.

Mini-batch is therefore more efficient than stochastic, due to vectorised implementation, and is more smooth compared to stochastic.

3b

If we initialise weights symmetrically, then all the units in the second layer will have the same value , and therefore all other layers will have the same value as well.
Therefore, each layer can be reduced to just a single unit, and this neural network is incapable of any real deep learning.
To counter this, we use random initialisation, so that all weights have random values. This is known as a **symmetry breaking** step.

3c

Regularization can reduce overfitting, because when we have an optimum value of lambda, it will reduce the weights that make the NN overfit.

However, having a large lambda would reduce all the weights to small values, and therefore makes the neural network very simple. On the other hand, having a very small lambda(i.e. no regularization) Would lead to overfitting.

Other methods to control overfitting are **dropout**, in which we randomly eliminate some neurons/weights on each iteration, based on a probability. Therefore it is almost as if we are working with a different smaller neural network each iteration, which makes it difficult for the neural network to overfit the data

3d

Batch normalization is the idea that instead of just the input layer, we normalise each layer of the neural network before performing the forward-propagation step. In practice, we usually normalise before applying the activation Function

Generally, SGD (Stochastic Gradient Descent) undoes the normalisation if it minimises the loss function. Hence batch normalisation adds two trainable parameters to each layer, the standard deviation(gamma) and the mean(beta) , and only allows SGD to change these parameters, rather than all the different weights in order to maintain the stability of the system.

There are several advantages to Batch Normalisation

- This greatly speeds up learning, and allows each layer of a network to learn slightly more independently of the other layers
- We can use higher learning rates, as batch normalisation makes sure that there is no activation that has gone really high or really low.
- It also reduces overfitting as it has slight regularization effects

4

Since the number of output channels is 8, we know the last dimension of the conv. layer is 8.
Now the height and width of the conv. layer is 3*3, and the number of input channels is 3. Hence the number of parameters is 3*3*3*8 = 216.

The image has dimensions N1, and N2. So the dimensions of the output is $\left(\lceil \frac{N1}{2} \rceil - 1\right) * \left(\lceil \frac{N2}{2} \rceil - 1\right) * 8$