OPTIMIZATION AND REGULARIZATION IN NEURAL NETWORKS

FALL SEMESTER 2018

PROF. ANKITA PRAMANIK

SILADITTYA MANNA

BIAS AND VARIANCE

First we need to know, what is overfitting and underfitting and how is bias and variance related to them.

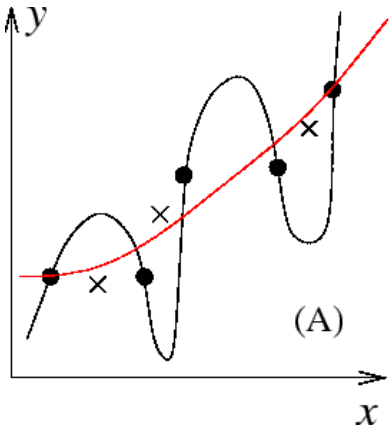
In statistics, fitting a function means how well you approximate a target function.

In machine learning, the target is to fit the data to an appropriate mapping function, so as to use it for future predictions.

Overfitting

Overfitting means that the model that you are training, fits the training data too well.

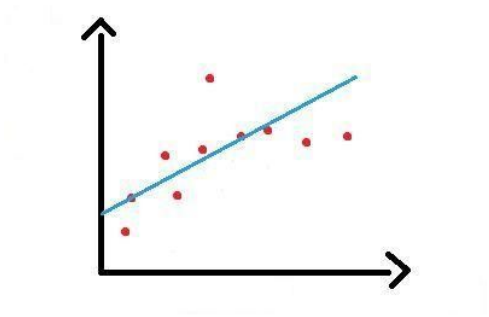
This happens when the model learns the intricate details of the training data as well as the noise. That is, it will learn the noise or randomness, as part of the original signal, underlying the data. This happens if the algorithm is too complex, or if it the regularization is not properly done.



Thus, the performance degrades. When a new data is given as input, the model fails to perform as good as it does on the training data, thereby, rendering the model unable to generalize to new data, because, when making predictions, the model also takes into account the features of the noise, it has learnt.

Underfitting

Underfitting occurs when the model fail to learn the features of the training data and also fails to generalize to new data. It has poor performance on both training and new data.



A statistical view of the Training Data

Let, the training set D for our neural network be D = $\{x_i, y_i : i = 1....n\}$

Now, the output yi for each input xi, will follow a regressive pattern, or say has a definite mapping between them, which is $y_i = g(x_i) + \varepsilon$, where ε is the random noise

Now, $g(x_i) = E[y|x_i]$, is the statistical expectation over all the training examples.

However, we can ignore such complexity and just consider y = g(x).

Bias

Bias in statistics is defined as the difference between the parameter to be estimated and the mathematical expectation of the estimator. In neural network, bias denotes the degree of fitting it achieves on the training data.

Bias = $E[\hat{y}] - y$

Variance

Variance in statistics is defined as expectation of the squared deviation of a variable from its mean. It denotes how far the values are spread out from the mean. In neural network, the variance denotes how well the neural network performs when provided with a new data.

Variance = $E[(\hat{y} - E[\hat{y}])^2]$

Suppose, ***y*** is the output for the input $\boldsymbol{x_i}$, and the model output is $\hat{y}$

So, ***y*** is the estimator, and $\hat{y}$ is the parameter to be estimated.

Hence, the mean of the squared error over all the dataset is

MSE = $E_D[(\hat{y} - y)^2] = E[\hat{y}^2] + y^2 - 2.E[\hat{y}].y$

Bias^2($\hat{y}$**,**y) = $(E[\hat{y}] - y)^2 = E^2[\hat{y}] + y^2 - 2.E[\hat{y}].y$

Variance($\hat{y}$) = $E[\hat{y}^2] - E^2[\hat{y}]$

Therefore, MSE = Bias^2($\hat{y}$ **,y**) + Variance($\hat{y}$)

So, now we can see that, for over-fitted model, the bias will be zero or very low, hence, the Variance will be high.

And for the under-fitted model, the opposite will be true.

Remedies of these problems

1. High Bias:
    a. Increase number of nodes or layers
    b. Train longer.
2. High Variance:
    a. Can be found out by looking at Validation Set performance
    b. More data
    c. More regularization
    d. Reducing the number nodes/layers

Bias-Variance tradeoff

A good supervised machine learning model will have low bias and low variance.

However, from the above decomposition of MSE into Bias and Variance, we can see that, if we increase Bias, Variance will decrease and vice-versa.

Thus, there is a trade-off between the bias and variance.

Regularization

L2 Regularization

Suppose, we have a logistic regression model, with the cost function **J(W,b),** where W and b are the parameters.

$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$,where $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$ is the L2 regularization

For a matrix, it is called the "Frobenius Norm" $(\|w\|_F^2)$ of the weight matrix.

L2 regularization is also called "Weight Decay"

$$dW^{[l]} = \frac{dJ}{dW^{[l]}} + \frac{\lambda}{m} W^{[l]}$$

$$W^{[l]} = W^{[l]} - \alpha\, dW^{[l]} = W^{[l]} - \alpha \frac{dJ}{dW^{[l]}} - \alpha\frac{\lambda}{m} W^{[l]} = (1 - \alpha\frac{\lambda}{m})W^{[l]} - \alpha\frac{dJ}{dW^{[l]}}$$

So, we have a new term in the update equation, where we are subtracting $\alpha\frac{\lambda}{m}$ from 1. Hence, the weights gets decayed, than the weights without it.

L1 Regularization

$$\frac{\lambda}{2m}\|w\|_1 = \frac{\lambda}{2m}\sum_{j=1}^{n_x} |w_j|$$

Using L1 regularization makes the Weights sparse.

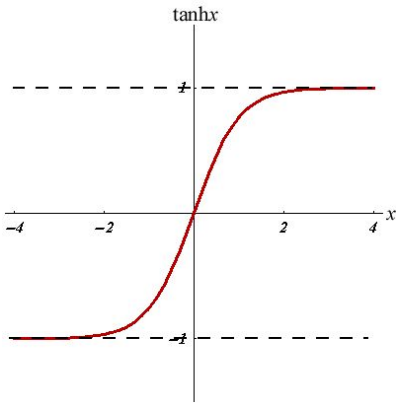
λ is called the Regularization parameter. It is also another Hyper-parameter.

Why L2 Regularization reduces over-fitting?

Adding the L2 norm or Frobenius norm, to the Cost function, penalizes the weight matrices from being too large.

Intuitive Interpretation: If λ is very very large, then weights become very very small, almost close to zero, and then a complex network can get effectively reduced to be equivalent to a smaller simpler network. And hence, a network with Low bias and High variance, may shift towards a network with High bias and Low variance.

Although the weights does not get reduced to zero, but to a very low value, where the effects of the weights become negligible, and thus the network gets reduced to a simpler smaller network.



Let us consider a tanh activation function.

Now, if λ is increased, then $W^{[l]}$ will decrease

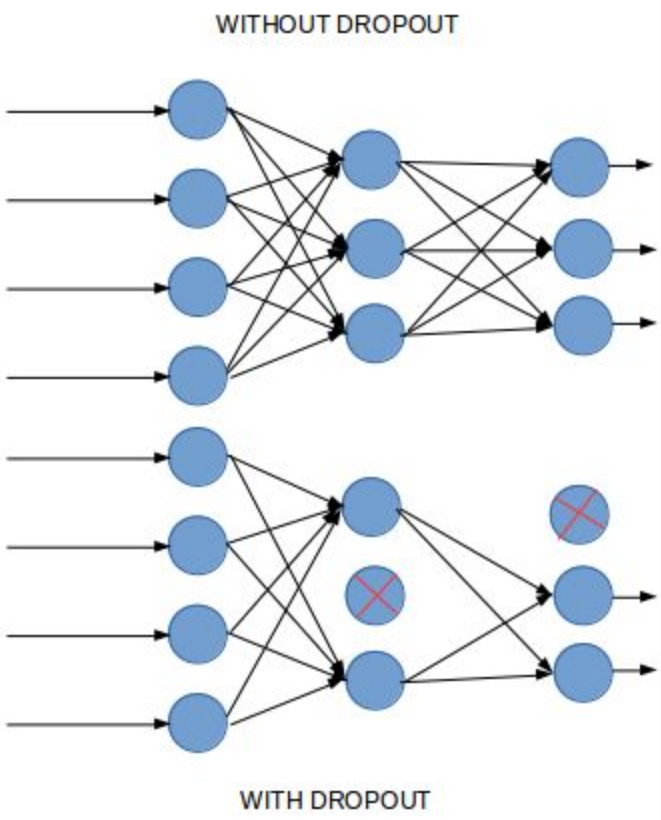
Then, $z^{[l]} = W^{[l]}.X + b^{[l]}$ will also be very less.

Thus, as can be seen from the tanh function graph, for low value of z, then tanh(z) will be roughly linear.

Thus, is all the layers are linear, then the whole network will be linear.

Hence, the network will not be able to able to fit the complex data as in the over-fitting case, and hence the bias will be more than in over-fitted condition.

Dropout Regularization



In dropout, we simply ignore some neurons during the training phase, i.e. these neurons are not considered during the forward and backward propagation passes.

However, they are considered during the testing phase.

Individual nodes are dropped out at probability 1-*p* or kept at probability *p*.

Dropout enables the network to learn more robust features.
But with dropout the convergence time increases, hence the network needs to be trained for longer period of time. However, since several nodes are removed, the training time for each epoch will be less.

Also, for N number of hidden units, there are 2^N possibilities, the nodes can be dropped, hence, we will have 2^N possible models.

Why does Drop-out works?

Individual neurons gets eliminated randomly. Hence, the network, cannot put too much significance on any one of the input weights. Hence, it will spread out the weights, which will shrink the squared norm of weights. Thus, it has the same effect as L2 regularization.

Other Regularization Methods:

Data Augmentation: Flipping, Cropping, Scaling, Rotating, Shearing, Brightness change, Contrast change.

Early Stopping: Sometimes, the Validation Set Error, increases after decreasing for a while.

Early stopping means stopping the training procedure of the network, before the validation error starts increasing. Because you have a moderate value of $||w||_F^2$ at this point and will again start increasing after this minimum point.

Normalizing Inputs

Normalizing Inputs corresponds to two steps:

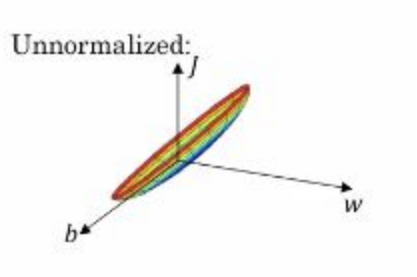
1. Subtracting the mean
2. Normalize by the Variance

$X = X - \mu$

$X = X / \sigma^2$

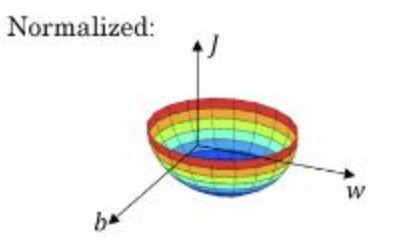
Why normalize the Input features?

Without normalization the Cost function will look like this.



If the feature vectors X_1 and X_2 takes on different values of different scales, then the weights w_1 and w_2, will also, take very different values.

However, if we normalize the input, the cost function will look like this



Thus, the error surface reduces from a elliptical surface to a circular circular one.

Thus, the gradient descent can proceed more smoothly, towards the minima, in the normalized surface, than on the unnormalized one.

Vanishing and Exploding Gradient

Suppose, you have a 20 layer deep network.

Considering $b^{[n]} = 0$ and g(z) = z

The output of the network is $\hat{y} = W^{[l]}W^{[l-1]}...W^{[3]}W^{[2]}W^{[1]}x$

Let, $W^{[l]} = 1.5\ I$

Hence, the product of the weight matrices of the successive layers, will become exponentially large.

$\hat{y} = 1.5^{20}.I.x$

Thus, the activation values will also be exponentially large.

The derivative of the activation functions, and hence the gradients of the weights will also be exponentially large. This is called exploding gradient.

If, $W^{[l]} = 0.5\, I$

Then, the output will be $\hat{y} = 0.5^{20}.I.x$

Thus, the output, and also the consecutive weights will also be exponentially small, or very small numbers.

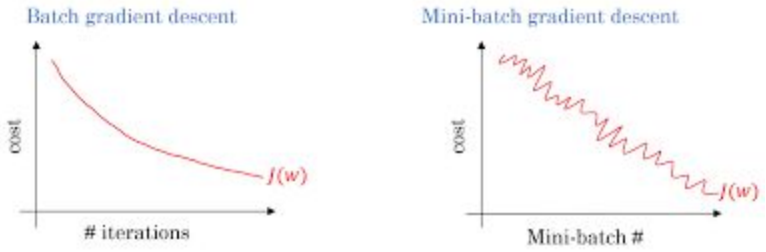
This, causes the gradients to also becomes very small. This problem is called Vanishing Gradient Problem.

This problem can be averted by using proper initialization. However, there are also other methods to prevent this.

Optimization Algorithms

Mini-batch GD

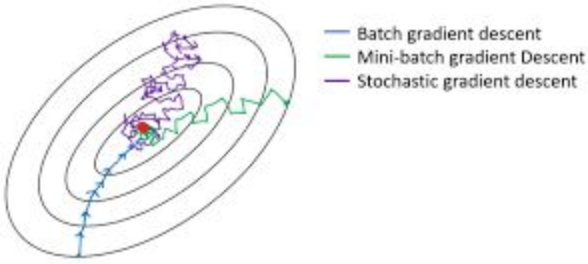
In GD, the whole training set is used to calculate the cost for one step. So, the updation of weights for a single step, takes too much time. Hence, the training set is split into smaller batches, and then each batch is used, instead of the whole training set. This makes the process faster and also more efficient. This is called Mini-Batch Gradient Descent.



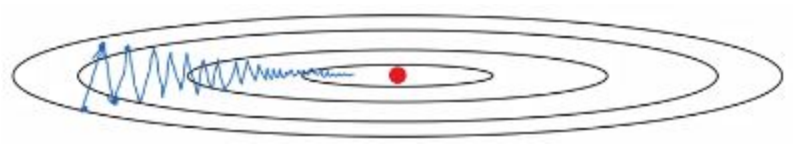
However, the cost function in MBGD, is not smooth as in Batch GD, but is a little noisy.

The most important factor is to select the batch size, which is also an important hyper-parameter. Large size of batch, tends to make the algorithm slower, whereas, using smaller values, makes it noisy, reducing the chances of convergence, and will hover around the minimum, but never really reaching the minimum.

If the batch is size is equal to 1, the it is called Stochastic Gradient Descent.



GD with Momentum



In, gradient descent, the successive steps are as given in the picture. There are several up and down oscillations, before it reaches the minima. So, there is a need to use different learning rate for different stages of the optimization process.

Thus, this prevents us from using larger learning rate, which may cause the gradient descent algorithm to shoot up.

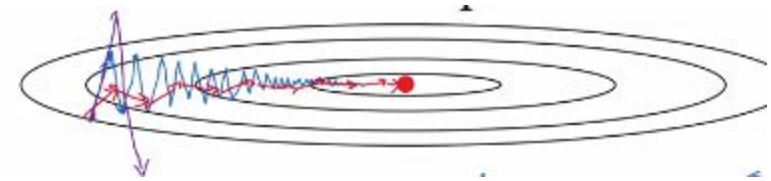
So, the objective is slower learning along the vertical direction, but faster learning along the horizontal direction.

By using **momentum**, we can smooth out the oscillations.

$$V_{dw} = \beta V_{dw} + (1-\beta)\, dw$$

This is essentially an **exponentially weighted average function**.

The update rule is: $W = W - \alpha V_{dw}$



Here, we get an extra hyper-parameter β along with learning rate α

RMSProp

In GD, we can end up with huge oscillations. So, to slow down the learning, what RMSProp does is,

$$S_{dw} = \beta S_{dw} + (1-\beta)\, dw^2$$

$$S_{db} = \beta S_{db} + (1-\beta)\, db^2$$

$$W = W - \alpha\, (dw/\sqrt{S_{dw}})$$

$$b = b - \alpha\, (db/\sqrt{S_{db}})$$

If the oscillation is high in the vertical direction, then db^2 is very large, hence the update for b is small, whereas the opposite is true for W.

To, make sure that the algorithm doesn't divide by zero, a small number ε is added to the denominator in the update step.

Adam

$$V_{dw} = \beta_1 V_{dw} + (1-\beta_1)\, dw$$

$$S_{dw} = \beta_2 S_{dw} + (1-\beta_2)\, dw^2$$

For bias correction

$$V_{dw}^{corrected} = V_{dw}/(1-\beta_1^{t})$$

$$S_{dw}^{corrected} = S_{dw}/(1-\beta_2^{t})$$

$$W = W - \alpha\, (V_{dw}^{corrected}/\sqrt{S_{dw}^{corrected}} + \varepsilon)$$

Batch Normalization

Batch normalization is a technique which eases the difficulty to train very deep models.

When calculating the gradient, it is assumed that, the other variables remain constant, but when the update are done, all the updates are made simultaneously.

Suppose, we have a network, with only one unit per layer and has activation function

g(z) = z

$\hat{y} = w_l.w_{l-1}.....w_2.w_1.x$

Here, w_i provides the weight used by the layer i.

The output of layer i is $g_i = g_{i-1}.w_i$

The output $\hat{y}$ is a linear function of the input x but a nonlinear function of the weights w_i .

Suppose, we have a gradient from back-propagation equal to ∇_w

So, we have to decrease the weights by this quantity.

So, the update is $w = w - \alpha.\nabla_w$

After updating all the weights

The new output will be $\hat{y} = (w_l - \alpha.\nabla_l)(w_{l-1} - \alpha.\nabla_{l-1}).....(w_2 - \alpha.\nabla_2)(w_1 - \alpha.\nabla_1)x$

Thus, there will be second and third order effects along with first order effects.

Thus the update of one layer depends on the other layers.

In very deep networks, even higher order interactions can be significant.

Batch normalization reduces this problem of higher order interactions between parameters updates in different layers.

Let ***H*** be a activation of the layer to normalize.

Replace ***H*** by $H' = (H - \mu)/\sigma$,where μ is the mean of each unit and σ is the standard deviation of each unit.

$$\mu = (1/n) \sum H_i$$

$$\sigma = \sqrt{[\delta + (1/n)\sum (H_i - \mu)^2]}$$

As said by Ian Goodfellow, in his book

Crucially, we back-propagate these operations for computing the mean and the standard deviation, and for applying them to ***H***. This means that the gradient will never propose an operation that acts simply to increase the standard deviation or mean of h_i; the normalization operations remove the effect of such an action and zero out its component in the gradient.

Now, again coming back to the example

$$\hat{y} = w_l.w_{l-1}.....w_2.w_1.x$$

Since the transformation from x to $\boldsymbol{h_{l-1}}$ is linear, thus is x is drawn from a gaussian distribution with zero mean and unit variance, the $\boldsymbol{h_{l-1}}$ will also come from a gaussian distribution, but with non-zero mean and also the variance will not be unit. Thus it has to be normalized, which restores the properties, as in x.

Thus $\hat{y} = w_l.h_{l-1}$ will be a linear relation, and hence can be learned easily.

The lower layers has no effect on $\boldsymbol{h_{l-1}}$, as they are always normalized to unit gaussian.

Normalizing the mean and standard deviation of a unit can reduce the expressive power of the neural network containing that unit. To maintain the expressive power of the network, it is common to replace the hidden unit activations ***H*** with $\gamma H' + \beta$. The variables γ and β are learned parameters that allow the new variable to have any mean and standard deviation. This new parameterization is better because, the mean is decided solely by β

and is easier to learn, whereas in the old parametrization the mean of ***H*** is determined by complicated interaction between the parameters in the layers below ***H***.

Most neural network layers take the form $\Phi(XW+b)$, where Φ is some fixed nonlinear activation function such as the rectified linear transformation. It is natural to wonder whether we should batch normalization to the input **X**, or to the transformed value **XW+b.** More specifically, **XW+b** should be replaced by a normalized version of **XW**. The bias term should be omitted because it becomes redundant with the β parameters applied by the batch normalization reparameterization. The input to a layer is usually the output of a nonlinear activation function such as the rectified linear function in a previous layer. The statistics of the input are thus more non-Gaussian and less amenable to standardization by linear operations.

Softmax Regression

Softmax Regression is an extension of Logistic Regression, rather a generalization, which allows us to handle multiclass regression problems.

In Softmax, the number of labels are {1,...,K} where K is the number of classes.

Given an input x, the objective is to estimate the probability that P(y = k|x) for each value of k = 1,....,K.

Thus, the output will be a K-dimensional vector, whose sum is 1.

$$P(y|x) = \frac{e^{W^T x+b}}{\sum_{k \varepsilon Y} e^{W^T x+b}}$$

The cost function is

$$J(W,b) = -\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)}=k\} log(P(y^{(i)}=k|x^{(i)};W,b))$$