LOSS

1) Equations for L1 loss : $L_i(\theta) = \sum_{j=1}^{K} |\hat{y}_j^{(i)} - y_j^{(i)}|$

Equation for L2 loss : $L_i(\theta) = \sum_{j=1}^{K} (\hat{y}_j^{(i)} - y_j^{(i)})^2$.

Equation for Huber loss : $\ell_\delta(d) = \begin{cases} \frac{1}{2}d^2 & \text{if } |d| \le \delta \\ \delta\left(d - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases}$

$$L_i(\theta) = \sum_{j=1}^{K} \ell_\delta\left(\hat{y}^{(i)} - y^{(i)}\right).$$

Equation for log-cosh loss : $L_i(\theta) = \sum_{j=1}^{K} \log\left(\cosh\left(\hat{y}^{(i)} - y^{(i)}\right)\right)$

where $\log(\cosh(d)) \approx \begin{cases} d^2/2 & \text{if } d \text{ is small} \\ |d| - \log(2) & \text{otherwise.} \end{cases}$

- L1 and L2 loss ~~tries~~ tries to minimizes distance between known and predicted values. If there is an outlier this will produce high loss even though our loss is not too much high without outliers.

- Huber loss handles outlier by, using quadratic ~~loss~~ function for small value of 'd' ($d \le \delta$) and linear function for bigger value of d to calculate loss.

- Log-cosh loss does the same job of handing the ~~outl~~ outlier ~~by using~~ by using quadratic function for small 'd' value and linear function for larger value.

2) Equation for cross entropy loss $= -\sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right)$

where $L_i(\theta) = -\sum_{j=1}^{K} y_j^{(i)} \log\left(\hat{y}_j^{(i)}\right)$ is sample loss.

Derivative for log-likelihood :-

Likelihood : $L(\theta) = \prod_{i=1}^{m} \prod_{j=1}^{k} \left(P\left(y=j \mid x^{(i)}\right)\right)^{y_j^{(i)}}$

Taking log on both hand side,

$$\log L(\theta) = \sum_{i=1}^{m} \sum_{j=1}^{k} y_j^{(i)} \log \left(P\left(y=j \mid x^{(i)}\right)\right)$$

- We are interested in maximizing log likelihood, which is equivalent to minimizing negative likelihood during training i.e.

Cross entropy loss, $L = -\log L(\theta)$

$$= -\sum_{i=1}^{m} \sum_{j=1}^{k} y_j^{(i)} \log \left(\hat{y}_j^{(i)}\right).$$

For random class assignment

$$P\left(y=j \mid x^{(i)}\right) = \frac{1}{K} \longrightarrow -\log\left(P\left(y=j \mid x^{(i)}\right)\right) = \log(K).$$

Hence the worst cross entropy loss value you except for random assignment

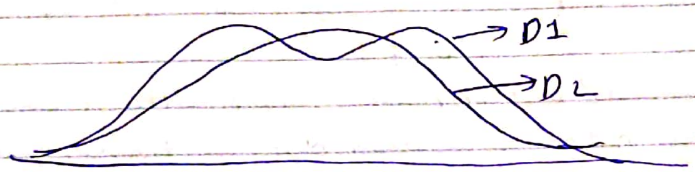
$$\sum_{i=1}^{m} \sum_{j=1}^{K} y_j^{(i)} \log K.$$

3) Softmax loss function → We use softmax as the activation function in the output layer and we use cross-entropy loss. Hence Softmax loss is same as cross entropy loss.

$$L = \sum_{i=1}^{m} \sum_{j=1}^{k} y_j^{(i)} \left(\hat{y}_j^{(i)}\right).$$

4) Equation for KL loss : $L = -\sum_{i=1}^{m}\sum_{j=1}^{k}\left(\frac{y_j^{(i)}}{\hat{y}_j^{(i)}}\right)$.

- KL divergence measures the similarity between two distributions.



Let take IID samples $\{x_i\}_{i=1}^{m}$ and distributions $P(x)$ and $q(x)$ that we want to compare.

Determine likelihood ratio for entire dataset.

$$\frac{P(x_1, x_2, \ldots, x_m)}{q(x_1, x_2, \ldots x_m)} = \prod_{i=1}^{m}\frac{P(x_i)}{q(x_i)} \begin{cases} >1 & P(x) \text{ is better} \\ <1 & q(x) \text{ is better} \end{cases}$$

Determine likelihood ratio for the entire dataset.

$$\sum_{i=1}^{m}\log\left(\frac{P(x_i)}{q(x_i)}\right) \begin{cases} >0 & P(x) \text{ better model} \\ <0 & q(x) \text{ better model.} \end{cases}$$

compare expected value of log likelihood ratio.

~~$E_{x\sim p}\left[\log \frac{p(x)}{q(x)}\right]$~~

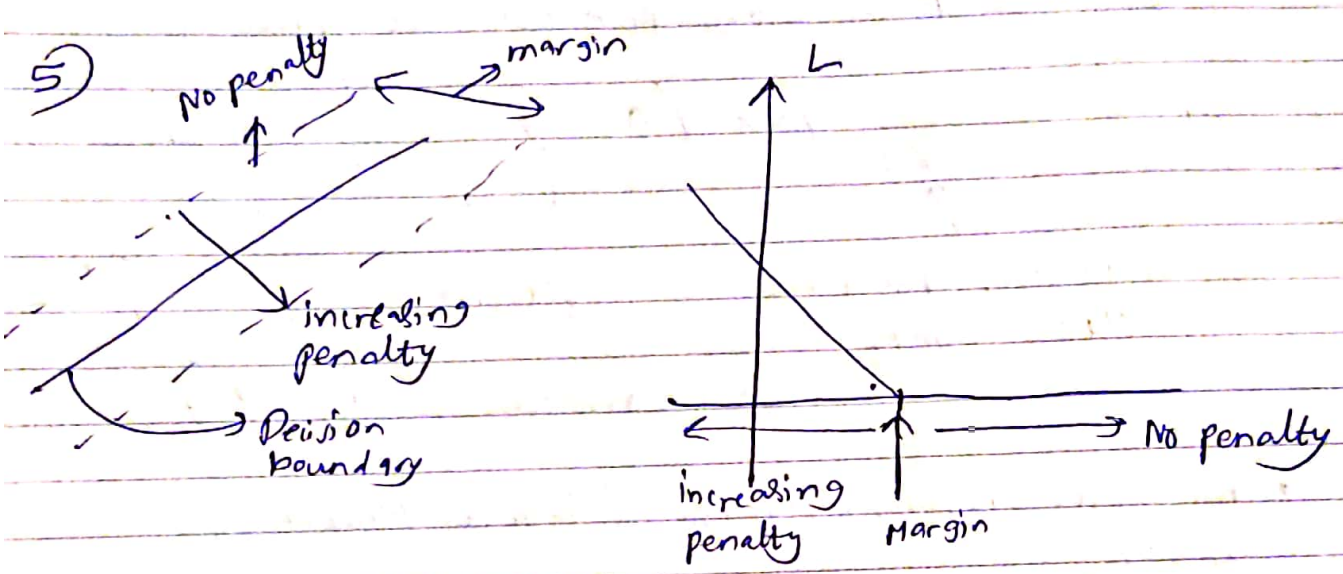
$$E_{x\sim p}\left[\log\frac{P(x)}{q(x)}\right] = \int_{-\infty}^{\infty} p(x)\log\frac{P(x)}{q(x)}\,dx.$$

$$\simeq \sum_{i=1}^{m} p(x_i)\log\frac{p(x_i)}{q(x_i)} = KL(P\|q).$$

When $KL(P\|q) \approx 0$, P and q are similar.

- In our case $P(x_i) = y^{(i)}$, $q(x_i) = \hat{y}^{(i)}$. Hence if $y^{(i)}$ does not change, using KL loss is equivalent to cross entropy ie. there is no difference between both.

5)



SVM Loss 2 class problem.
No penalty on the correct side of the margin and increasing penalty on the incorrect side and it increases as we move away from decision boundary in the opposite direction as that of the correct class.

Assume a 2-class problem with correct labels $y^{(i)} \in \{+1, -1\}$ and classifier score $\hat{y}^{(i)}$. We take margin of 1.

Hinge loss

$$L_i(\theta) = \max(0, 1 - y^{(i)} \hat{y}^{(i)})$$

→ positive if predictions agree in sign with margin < 1 or disagree in sign
→ Negative IF prediction agree in sign with margin > 1.

- Similarity extending SVM loss to a K-class problem:
  $y^{(i)} \in [1, K]$: true label for $x^{(i)}$.

  $\hat{y}^{(i)} \in R$ = Score prediction for class j.

$$L_i(\theta) = \sum_{j \neq t} \max(0, \hat{y}_j^{(i)} - \hat{y}_t^{(i)} + 1) \quad \text{where } t = y^{(i)}.$$

Sum over incorrect class labels

positive if $\hat{y}_t^{(i)} > \hat{y}_j^{(i)} - 1$ (incorrect)

negative if $\hat{y}_t^{(i)} < \hat{y}_j^{(i)} - 1$ (correct)

- for low random scores (before learning)

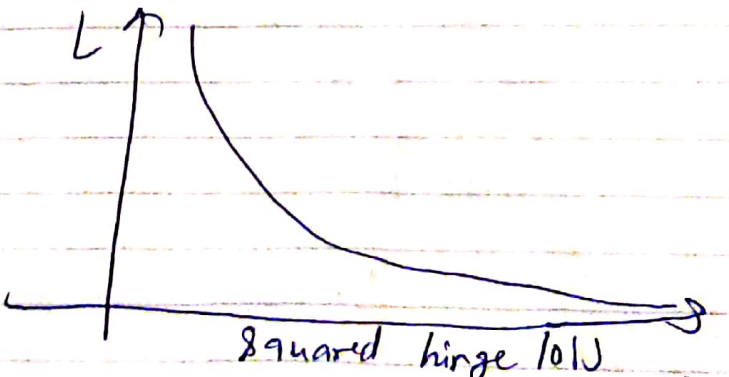
  $\hat{y}_j^{(i)} \approx 0 \rightarrow L_i = \max(0, 0 - 0 + 1) + \cdots + \max(0, 0 - 0 + 1)$

  $= (K-1) \times 1$

  $= (K-1) \leftarrow$ worst value.

Squared hinge loss

$$L_i(\theta) = \frac{1}{2} \sum_{j=1}^{K} \max(0, \hat{y}_j^{(i)} - \hat{y}_t^{(i)} + 1)^2.$$



	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
θ			
$\hat{y}_1$	0.5	1.3	1.5
$\hat{y}_2$	0.4	-0.8	-0.4
$\hat{y}_3$	0.33	-0.6	2.7
y	1	2	3

$L1 = \max(0, 0.4 - 0.5 + 1) + \max(0, 0.3 - 0.5 + 1)$
$= 0.9 + 0.8 = 1.7$

$L2 = \max(0, 1.3 - 0.8 + 1) + \max(0, -0.6 - 0.8 + 1)$
$= 1.5 + 0 = 1.5$

$L3 = \max(0, 1.4 - 2.7 + 1) + \max(0 - 0.4 - 2.7 + 1)$
$= 0 + 0 = 0$

7) According to Occam's Razor principle:
Simple explanations are better.
A simple solution is when the weights (θ) are lower. Smaller weights results in more stable solution that will generalize better. That is the reason of using regularization.

- L1 regularization : $R(\theta) = \sum_{i,j} \theta_{ij}$

L2 regularization : $R(\theta) = \sum_{i,j} \theta_{ij}^2$.

L1 regularization makes weights sparse (concrete weights).
e.g. $|0.5| + |0.5| = |1| + |0|$.
L2 regularization makes weights smaller while spreading them e.g. $0.5^2 + 0.5^2 < 1^2 + 0^2$.

- $L(\theta) = \frac{1}{m} \sum_{i=1}^{m} L_i(f(x_i, w), y_i)) + \lambda R(\theta)$.

Regularization term coefficient i.e. λ determine how much importance we want to give to the regularization term. Too high value may result into underfitting of a model, too low value may result into overfitting of the model. λ is the hyperparameter that we need to ~~[illegible]~~ tune and choose based upon validation set performance.

8) Adding regularization term to loss results in weight decay.

$$L(\theta) = \frac{1}{m} \sum_{i=1}^{m} L_i \left(f(x^{(i)}, \theta), y^{(i)} \right) + \lambda R(\theta).$$

$$\frac{\partial L}{\partial \theta} = \frac{\partial}{\partial \theta}(a) + \frac{\partial}{\partial \theta}(\lambda R(\theta)).$$

For L2 : $\lambda \frac{\partial}{\partial \theta} R(\theta) = 2\theta\lambda$.

For L1 : $\lambda \frac{\partial}{\partial \theta} R(\theta) = \lambda \, \text{Sign}(\theta)$.

Hence during gradient descent for L2 loss subtract $2\theta\lambda$ from $\frac{\partial L}{\partial \theta}$ which is equivalent to weight decay even if gradient w.r.t. θ is 0.

For L1 regularization, this equivalent to subtracting $\lambda \sin(\theta)$ from derivation ~~of~~ of L w.r.t. θ.

9) Kernel regularization : regularize $w \rightarrow$ reduce w.
Bias regularization : regularize $b \rightarrow$ reduce b.
Activity regularization : regularize $y \rightarrow$ reduce y (and so w and b).

Normally only kernel regularization is used.
But sometimes bias regularization is used if function is expected to output small value.
In some other rare occasions we used activity regularization if function expected to output value close to zero.

Optimization :-

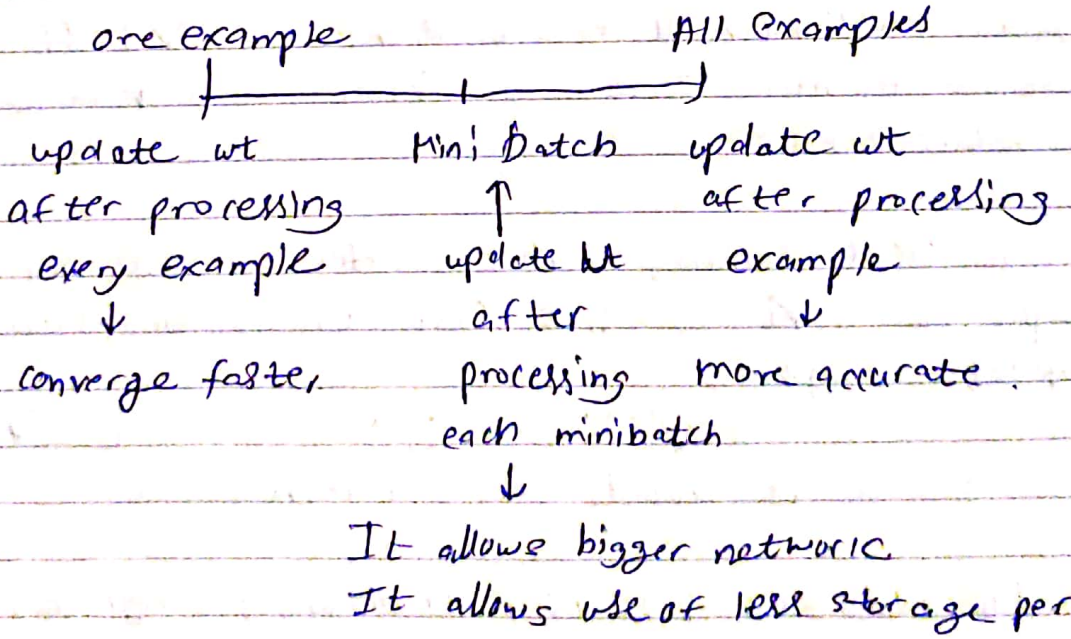
1) • Direct numerical computation of gradient is slow. Computation of gradient using backpropagation is fast and easy since in this process we break the complex derivative into simple ones using chain ~~rule~~ rule.

• Direct numerical ~~comp~~ computation of gradient can be used for verification of computed gradient on simplified network.

2) During updation of weight SGD uses only one example (current value) while in gradient descent we have to use all the examples to update weights.

• Since SGD uses only one example to update the weights, convergence is faster in SGD compared to GD.

3)

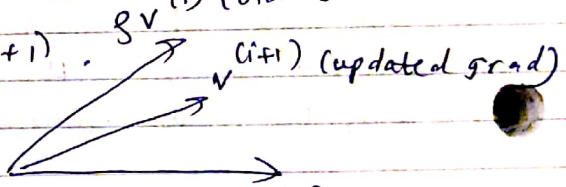


GD using one example converge faster while using all the examples & results in much accurate results. GD using mini batch allows to have the bigger network and less storage to process the data as compare to using all the examples.

- Problems with SGD
  i) What should be the learning rate.
  ii) What happens if loss is more sensitive to one parameter i.e. if its too faster in some directions and too slow in other.
  iii) How to avoid getting stuck in local minima.
  iv) Minibatch gradient descent are noisy.

4) SGD with momentum:
It smoothes out changes to gradient using momentum.

$$\begin{cases} v^{(i+1)} \leftarrow \rho v^{(i)} + \nabla L(\theta^{(i)}) \\ \theta^{(i+1)} \leftarrow \theta^{(i)} - \eta v^{(i+1)} \end{cases}$$

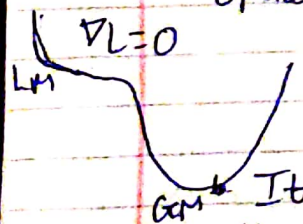


- Since it take the average of previous gradient and new gradient, it results in more stable and hence it reduces noise.

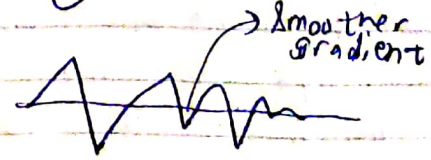
- In local minima, $\nabla L = 0$, Hence SGD without momentum stops at $\nabla L = 0$ but SGD with momentum continue to update weight,

$$v^{(i+1)} \leftarrow \rho v^{(i)} + 0 \quad [\because \nabla L = 0]$$

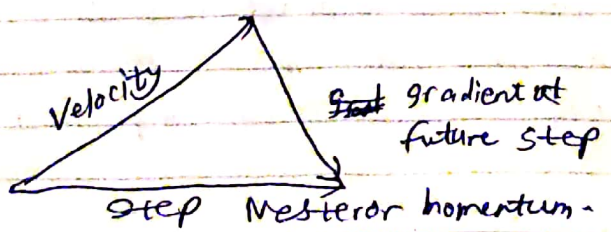
Hence it avoids local minima.

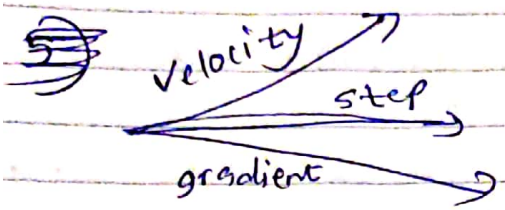


- It addresses the poor conditioning (high sensitive in some direction) by smoothing out by averaging with previous gradients.



5)





SGD with momentum.

Nestror momentum results in more accurate step of gradient compared to SGP + momentum, since it calculate gradient at a future step at the current point and then take step in direction of vector sum of velocity and gradient calculated at the future step.

- Writing Nesterove accelerated gradient equation:

$$\theta^{(i+1)} \leftarrow \theta^{(i)} + v^{(i)} + \beta\left(v^{(i+1)} - v^{(i)}\right).$$

The term $v^{(i+1)} - v^{(i)}$ is the difference between new velocity and old velocity. This term is called acceleration. Hence this is called Nesterove Accelerated Gradient.

6) Strategies for learning rate decay:-

(i) step decay:-
for every k iterations, $\eta \leftarrow \eta/2$

(ii) Exponential decay:
$$\eta = \eta_0 e^{-kt}$$
where k is decay rate and t is iteration index.

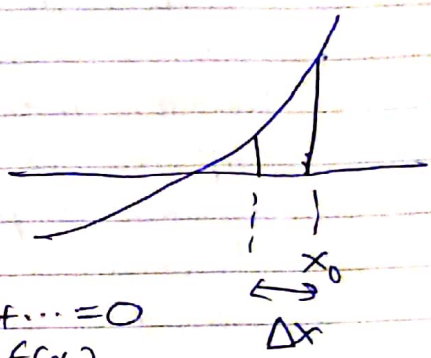
(iii) fractional decay:
$$\eta = \eta_0 / (1 + kt)$$
where k is decay rate and t is iteration index

7) Newton method to compute learning rate :-

Goal : compute learning rate instead of specifying it.

Newton's Algorithm :-

- Find x such that $f(x) = 0$
- Start with guess x_0
- find update Δx such that $f(x_0 + \Delta x) = 0$



- Taylor series expansion (linearization)

$f(x_0 + \Delta x) = f(x_0) + \Delta x f'(x_0) + \cdots = 0$

$\Rightarrow f(x_0) + \Delta x f'(x_0) = 0 \Rightarrow \Delta x = -\frac{f(x_0)}{f'(x_0)}$

- Continue while $f(x_0 + \Delta x) \neq 0$

Our problem :

$\nabla J(\theta) = 0$

$f(x)$

$f(x_0 + \Delta x) = f(x_0) + \Delta x^T \nabla f(x_0) = 0$

Replacing $f(x_0)$ with $\nabla J(\theta_0)$:

$\nabla J(\theta_0) + \nabla (\nabla J(\theta)) (\Delta \theta) = 0$

$H \leftarrow$ Hessian matrix.

$\nabla J(\theta_0) + H \Delta \theta = 0$

$\Delta \theta = H^{-1} \nabla J(\theta_0)$

Newton's method :

$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta H^{-1} \nabla J(\theta^{(i)})$

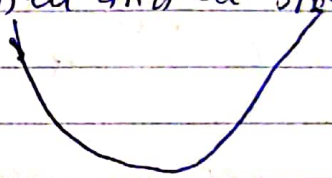
- Interpretation :-
  - Assume that H is diagonal

multiplied with $\nabla J(\theta)$.

$$H = \begin{bmatrix} h_1 & & \\ & \ddots & \\ & & h_n \end{bmatrix} \Rightarrow H^{-1} = \begin{bmatrix} 1/h_1 & & \\ & \ddots & \\ & & 1/h_2 \end{bmatrix}$$

where $h_i = \frac{\partial^2 J}{\partial \theta_i^2}$ = curvature

At high curvature, learning rate is small and at small curvature, learning rate is large.



8) Condition number :-

More generally we detect sensitivity using the singular values of the Hessian matrix :

Ratio of singular values of H w.r.t. smallest singular value.

Condition number : $\frac{sv_1}{sv_2}$ ← largest sv / ← smallest sv

- The problem is more difficult when high condition number.

9) Hessian computation is ~~expert~~ expensive and may be noisy. Hence we replace H with a different pre condition.

$$B^{(i)} = \text{diag}\left(\frac{1}{j=1} \sum \nabla J_j(\theta^{(i)}) \nabla j(\theta^{(i)})^T \right)^{1/2}.$$

Adagrad approximation of Hessian matrix.

$$B^{(i)} = \begin{bmatrix} \sqrt{\sum \left(\frac{\partial J}{\partial \theta_1}\right)^2} & & \\ & \ddots & \\ & & \sqrt{\sum \left(\frac{\partial J}{\partial \theta_n}\right)^2} \end{bmatrix}$$

$$B^{(i)^{-1}} = \begin{bmatrix} \frac{1}{\sqrt{\sum \left(\frac{\partial J}{\partial \theta_1}\right)^2}} & & \\ & \ddots & \\ & & \frac{1}{\sqrt{\sum \left(\frac{\partial J}{\partial \theta_n}\right)^2}} \end{bmatrix}.$$

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta B^{(i)^{-1}} \nabla J(\theta^{(i)}).$$

Large squared sum of past gradients in a particular direction means we use smaller steps in these directions. (Hessian Approximation).

10) In Adagrad, because we normalize by elementwise sum of square gradients, the step size will become smaller as iterations progresses.

To control this we use use in RMSprop a decay factor (0.9) when adding new gradient to the gradient sum.

$$s_j^{(i+1)} = \gamma s_j^{(i)} + (1-\gamma) \left\| \nabla_j L(\theta^{(i)}) \right\|^2.$$

$$\theta_j^{(i+1)} \leftarrow \theta_j^{(i)} - \eta \nabla_j L(\theta^{(i)}) \cdot \frac{1}{s_j^{(i+1)} + \epsilon}$$

where γ is decay factor.

11) Adam combines RMS prop (Scale by Sum of gradient elements) with momentum.

First moment : Velocity with momentum.

$$m_1^{(i+1)} = \beta_1 m_1^{(i)} + (1-\beta_1) \nabla L(\theta^{(i)}).$$

Second moment : elementwise & stepwise scale

$$m_2^{(i+1)} = \beta_2 m_2^{(i)} + (1-\beta_2)(\nabla L \theta^{(i)}) \odot \nabla L(\theta^{(i)}))$$

Combining momentum and elementwise step scaling

$$\theta^{(i+1)} = \theta^{(i)} - \eta \hat{m}_1^{(i+1)} \odot \frac{1}{\sqrt{m_2^{(i+1)} + \epsilon}}.$$

The above procedure fails because we do not know how to initialise m_1 and m_2.

If we take $m_1^{(0)} = m_2^{(0)} = 0$, then $\frac{1}{\sqrt{0+\epsilon}}$ and since ϵ is very small we give very large scale factor which result in very large step. This is the reason we use bias.

12) • Gradient Descent with line search

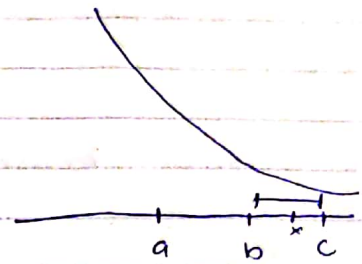
Instead of a fixed step size, find the 'best' step size.

- Given a direction
  $u = \nabla f(x)$
- Best step size:
  $\eta^* = \underset{\eta}{\operatorname{argmin}}\ f(x + \eta u)$

- Gradient descent:
  $\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta^{(i)} \nabla f(\theta^{(i)})$ ← η^*

To solve for ~~η^*~~ η^* : find explicit computation or use gradient descent (expensive) or perform simple line search :

Bracking Algorithm to find ~~η^*~~ η^*



Given bracket $[a, b, c]$

$x = \frac{b+c}{2}$

if $f(x) \leq f(b) \Rightarrow [b, x, c]$

if $f(x) > f(b) \Rightarrow [a, b, x]$.

continue with smaller and smaller brackets until the brackets is small enough.

Alternative to Bracking :

Alternative way to find optimal η step size is use gradient descent. But this is very costly since we have to find another gradient descent algorithm to find η inside gradient descent to find optimal θ_0.

13) Quasi - Newton Methods :

Algorithm:

1) $\Delta\theta := (H^{(i-1)})^{-1} \nabla J(\theta^{(i-1)})$.

2) Determine step size η^* (e.g. bracking line search)

3) Compute parameter updates

$\theta^{(i)} = \theta^{(i+1)} + \eta^* \Delta\theta$.

- Instead of computing updated Hessian approximate it is possible to compute updated $(H^{(k)})^{-1}$.

BFGS algorithm approximate the Hessian inverse using gradient evaluations whereas in Newton's methods we have to exactly calculate Hessian matrix which is very expensive. Hessian matrix has n^2 elements and inversting Hessian matrix has $O(n^3)$ complexity.

- Adam method tries to "imitate" Hessian matrix which mean its values does not correspond to values around Hessian matrix, On the other hand BFGS tries to approximate Hessian matrix which results in accurate scaling in gradient descent.

Regularization

1) Weight decay specified regularization in the neural network. During training, we add a regularization term to the loss value to compute the backpropagation derivatives. The weight decay value determines how dominant the regularization will be in the computation of gradient.

Therefore the weight ~~deg~~ decay loss is added to loss value multiplied by the coefficient λ.

$L_R(\theta) = L(\theta) + \lambda R(\theta)$ where $L(\theta)$ is network loss and $R(\theta)$ is weight decay.

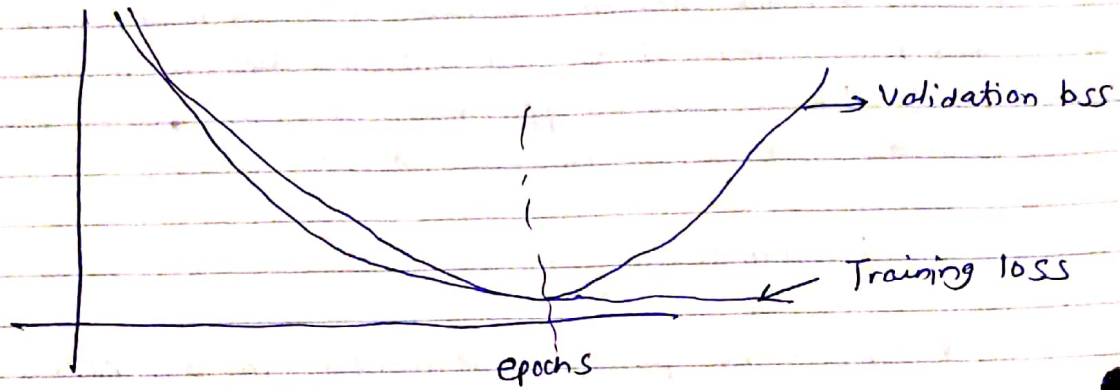
$$\frac{\partial L_R}{\partial \theta} = \frac{\partial L}{\partial \theta} + \lambda \frac{\partial R(\theta)}{\partial \theta}$$

for L_2 : $\frac{\partial R}{\partial \theta} = 2\theta$, during gradient we subtract $2\lambda\theta$

for L1 : $\frac{\partial R}{\partial \theta} = \lambda (\text{sing}(\theta))$, during gradient descent we subtract $\lambda \sin(\theta)$.

This is equivalent to adding a regularization to loss functions.

2) During training the network, we observe that after some iteration validation loss starts to increase but training loss continue to decrease. This suggests that, after this point our model starts to overfit. Hence to come up with a model with better generalization we truly train the model when the validation loss is minimum. Thus preventing overfitting.



Strategies to reuse the data :-

1) We retrain on all the data for a specific number of iterations determined from validation.

2) In this strategy, we continue training from previous weights with the entire data while validation loss is greater than training loss.

3) Data Augmentation is a regularization method which tries to generalize the model to prevent overfitting and basic stability to the model. In data augmentation, we augment the existing training data by performing various transformations.

In image classification problem, data augmentation is used to transform the existing data by performing transformations like rotating, scaling, cropping, adding noise etc. When we do such transformation, it helps the model generalize better. i.e. it helps ~~overfi~~ in preventing overfitting on training data ~~and~~ and result into better performance on unseen data.

By performing such transformation, we are adding variability to the training data.

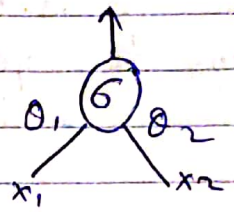
- We can augment data in feature or data domain.

4) Dropout refers to the process of ignoring units (neurons) in a fully connected network whose each unit is dropped with probability of $(1-p)$ or considered with a probability of p.

For a simple neurons with two inputs x_1 and x_2 and a sigmoid activation.

$$\hat{y} = E_D [f(x, D)]$$

$$= \int p(D) f(x, D) dD$$



$$\hat{y} = \frac{1}{4} 6(0 \cdot x_1 + 0 x_2) + \frac{1}{4} 6(0 \cdot x_1 + 1 \cdot x_2) + \frac{1}{4} 6(1 \cdot x_1 + 0 \cdot x_2) + \frac{1}{4} 6(1 \cdot x_1 + 1 \cdot x_2).$$

Multiplying the output of each node p is equivalent to computing expected value for 2^n dropped out networks.

Advantages of Dropout :-

1) Reduces dependency on a single node.
2) Distributes across multiple nodes.
3) Reduces overfitting
4) Reduces nodes interactions and increasing training speed.

Disadvantages :-

1) Longer training due to dropout, as all units are unavailable at each step.
2) Finding the value of 'p' is difficult sometimes.

5) To approximate expected values of all combinations of all dropped out network ~~efficiently~~ efficiently, we need to take a different approach of applying the dropout during training. Multiplying the output of each node by 'p' is equivalent of computing expected values for 2^n dropped-out networks.

$$\hat{y} = E_D [f(x, D)] = \int p(D) f(x, D) dD$$

↑ mask for all n nodes.

Instead of multiplying output by 'p' during prediction, it is possible to multiply outputs by 1/p during training. This helps in reducing the operations during predictions. So during testing and predictions runtime will be small.

6) Batch Normalization is a technique for improving speed, performance and stability. It is used to normalize the input layer by adjusting and scaling activations.

In a ~~the~~ neural network, batch normalization is achieved through a normalization step that fix the mean and variance of each layer's input. Ideally the normalization would be conducting over the entire training set, but normalization is retrained each mini-batch in the training process.

Batch outputs $\{z^{(i)}\}_{i=1}^{q} \rightarrow \{\hat{z}^{(i)}\}_{j=1}^{q}$

for current layer

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sigma_j}$$

↑ output of j^{th} unit for i^{th} batch example

$$\mu_j = \frac{1}{q} \sum_{i=1}^{m} z_j^{(i)}$$

$$\sigma_j = \left(\frac{1}{q} \sum_{i=1}^{m} (z_j^{(i)} - \mu_j)^2 \right)^{1/2}$$

During training, the batches are random and batch normalization adds randomness into the training thus reducing overfitting
But during prediction, we use the average normalization values computed during training.

- Batch normalization adds randomness to training because, the batches are selected at random, normalized and input into the network. This is the kind of randomness introduced into mainly ~~[illegible]~~.

7) Simply normalizing each input of a layer value may change what the layer can represent. Normalizing the inputs of a sigmoid would constrain them to the linear regime of the non-linear. To address this, we make sure the transformation added to the network to represent the identity transform.

To achieve this, we introduce two parameters $\gamma^{(k)}$ and $\beta^{(k)}$ which scale and shift the normalized values:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)}$$

These parameters are learned along the way and this process is called $BN_{\gamma,\beta} : x_1, x_2 \ldots x_n$

$\rightarrow y_1, \ldots y_n$ or batch normalizing transform.

The γ_k and β_k are learned and the network can learn to cancel BN transformation if there is no need for it.

$$z_j^{(i)} = \sigma_j \hat{z}_j^{(i)} + \mu_j = \sigma_j \frac{z_j^{(i)} - \mu_j}{\sigma_j} + \mu_j = z_j^{(i)}.$$

$$\gamma_j = \sigma_j, \quad \beta_j = \mu_j$$

8) Ensemble classifiers is a regularization method where we train multiple independent models i.e. its a way of generating various base classifiers from which a new classifier is derived which performs better than any other constituent ~~the~~ classifier. The base classifier might differ in the algorithm used, ~~by~~ hyperparameter, representation and training sets.

By doing the above ~~we~~ mentioned process, we can achieve better generalization by having different classifiers to select from. By doing this we can avoid overfitting as we try to look at the classifier with better results.

Strategies for producing esemble classifiers :-

Below mentioned strategies are used for creating ensemble classifier with multiple models.

1. Modified or different data set.
2. Different hyperparameter for ~~#~~ different models.
3. Record multiple snapshots of the model under different setting.