CS 577 - F22 - Assignment 3 Writing Portion

Due date: 10/25/2022

Anas Puthawala - A20416308
Professor Gady Agam

Loss

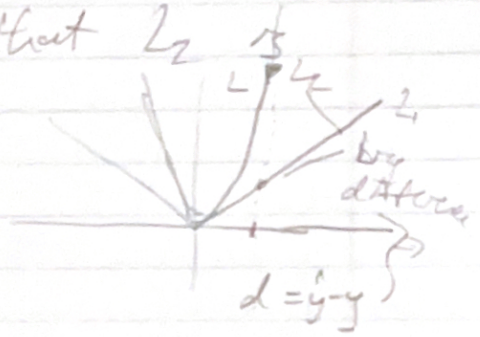
1. **L_1 loss** - regression problems, minimize distance between known & pred. values

$$L_i(\theta) = \sum_{j=1}^{k} \left| \hat{y}_j^{(i)} - y_j^{(i)} \right|$$

L_2 Loss - same scenario as above, except it's squared distance

$$L_i(\theta) = \sum_{j=1}^{k} \left(\hat{y}_j^{(i)} - y_j^{(i)} \right)^2$$

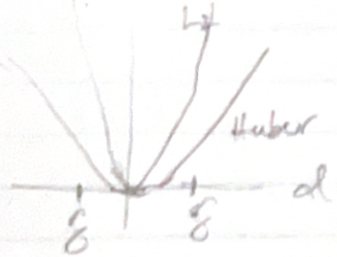
difference between L_1 & L_2 is that L_2 more sensitive to outliers



Huber loss - between L_1 & L_2 loss, it caps the loss of outliers

$$\mathcal{L}_\delta(d) = \begin{cases} \frac{1}{2}d^2 & \text{if } |d| \le \delta \\ \delta(d - \frac{1}{2}\delta) & \text{otherwise} \end{cases}$$

& quadratic for small d, linear for large d



$$L_i(\theta) = \sum_{j=1}^{k} \mathcal{L}_\delta(\hat{y}_j^{(i)} - y_j^{(i)})$$

k = # outputs

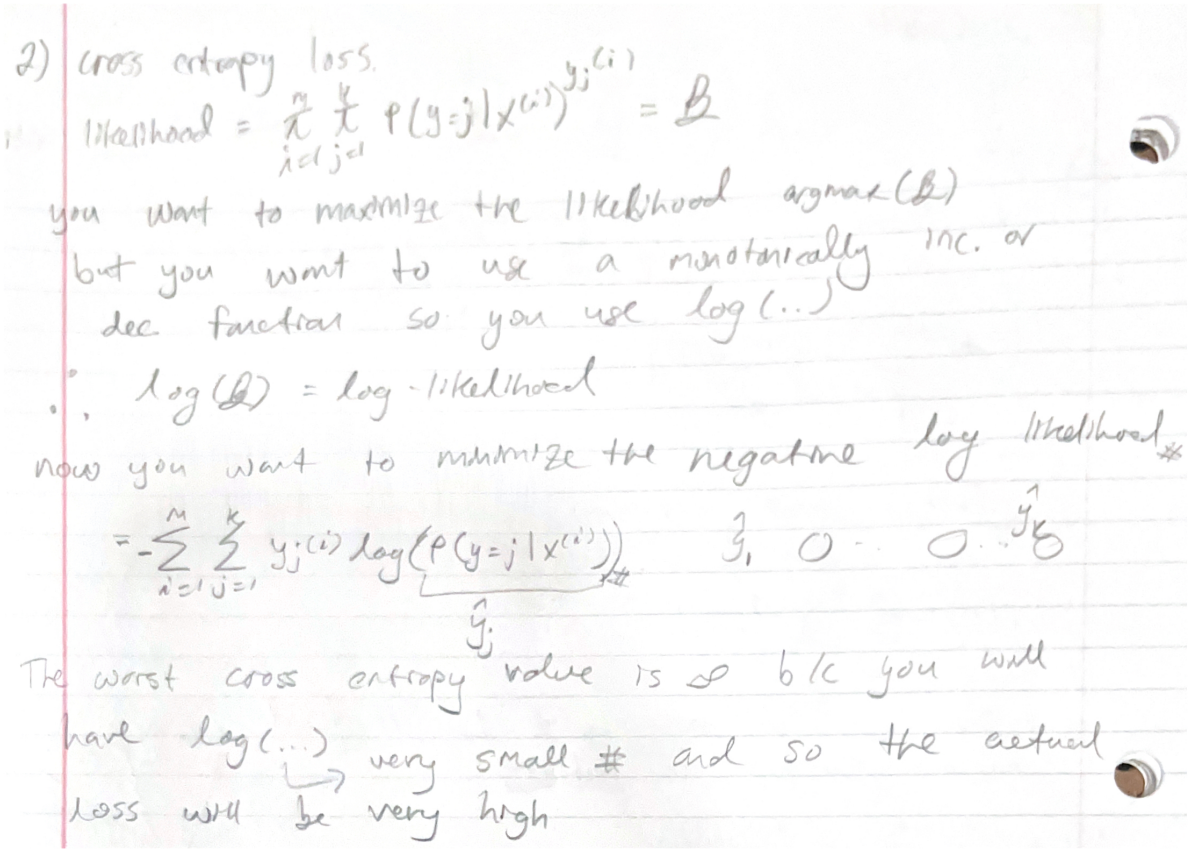
Log-Cosh loss - similar to huber loss

$$L_i(\theta) = \sum_{j=1}^{k} \log\left(\cosh\left(\hat{y}^{(i)} - y^{(i)}\right)\right) \qquad \log(\cosh(d)) = \begin{cases} \frac{d^2}{2} & \text{if } d \text{ is small} \\ |d| - \log(2) & \text{otherwise} \end{cases}$$

reduce sensitivity to outliers

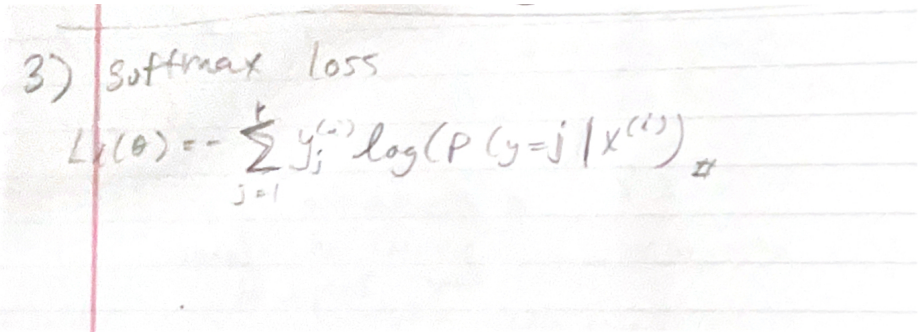
1) The main purpose of these loss functions as mentioned above are for regression tasks. They all differ in terms of sensitivity to outliers like for example L1 loss is a lot less sensitive to outliers as opposed to L2 loss. Huber, log-cosh loss are derivatives of the same type of loss function in terms of differing sensitivities to outliers. Each of them can be configured to the task at hand and the data at hand (i.e. choose a delta to properly penalize the outliers / data distribution) to improve upon learning.

2)



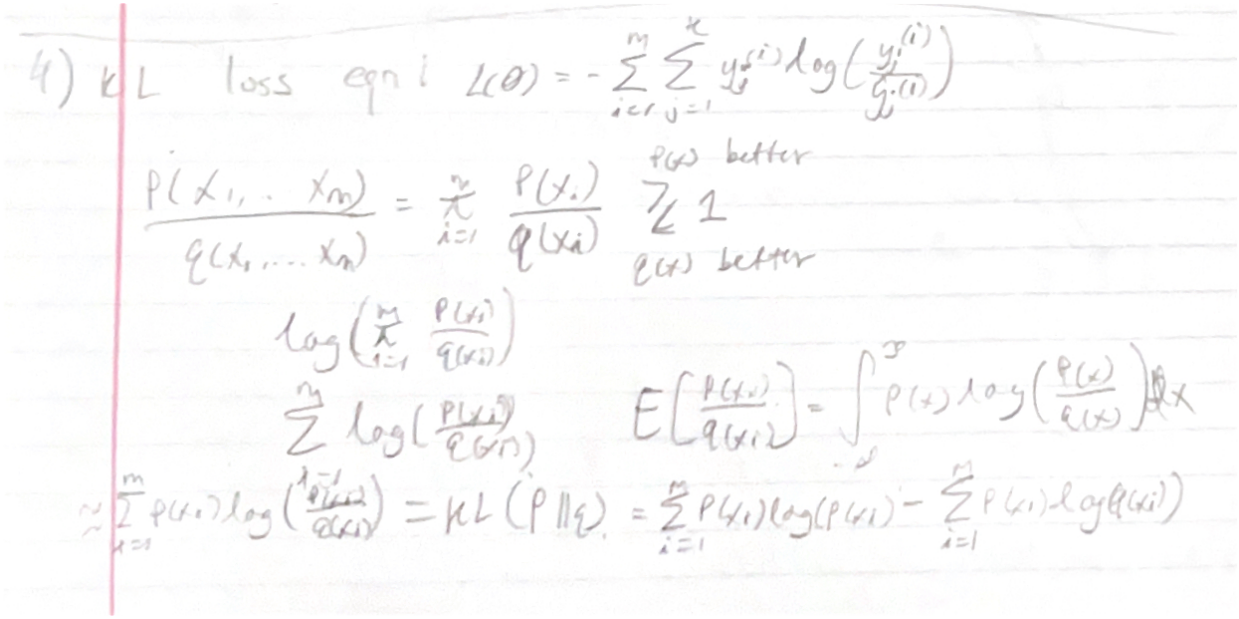
Additionally, when the classifier is really bad then it'll just output the same value for each of the units and in that case the output will be log(k).

3)

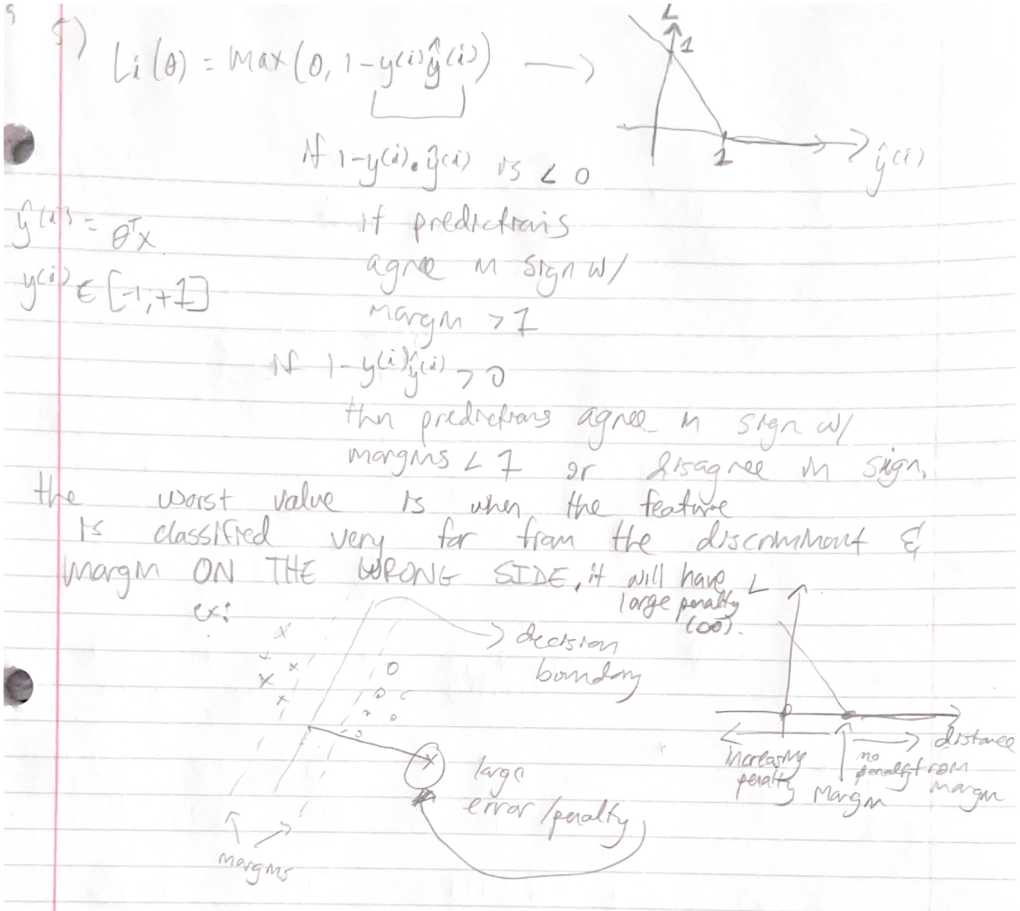


You would use softmax loss when dealing with multi-class classification problems. It turns the output vector into a probability that it belongs to a class in the class list of j=1 to K

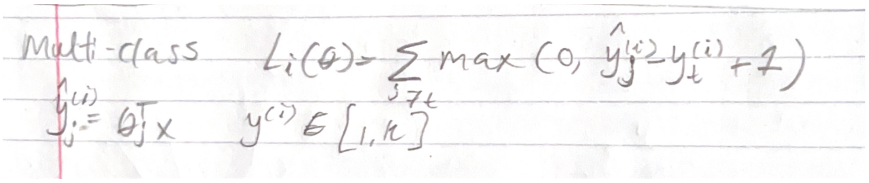
4) The KL loss measures the difference between two probability distributions. The KL divergence is basically the expected value of the log of the likelihood ratio of the two distributions p and q. If p and q are the same we have the minimum value, 0 (no error). The KL divergence is the same as cross-entropy when the class labels do not change. In other words, you can expand the KL loss to have two main terms; negative entropy and cross entropy and if you assume that the negative entropy is always the same since the dataset doesn't change then the loss is effectively identical to the cross entropy



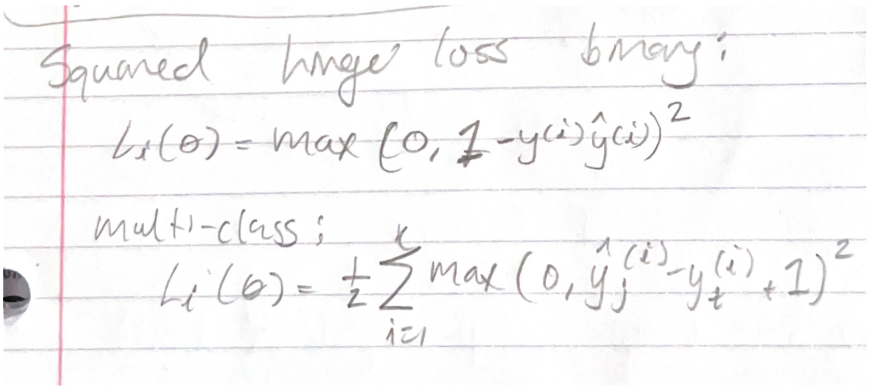
5) The motivation behind hinge loss is to have the neurons delineate a margin between the two classes. In other words, you want the discriminant to be of better certainty due to margins on either side. The y_i hat is the distance from the decision boundary (no longer a probability as in the previous losses)

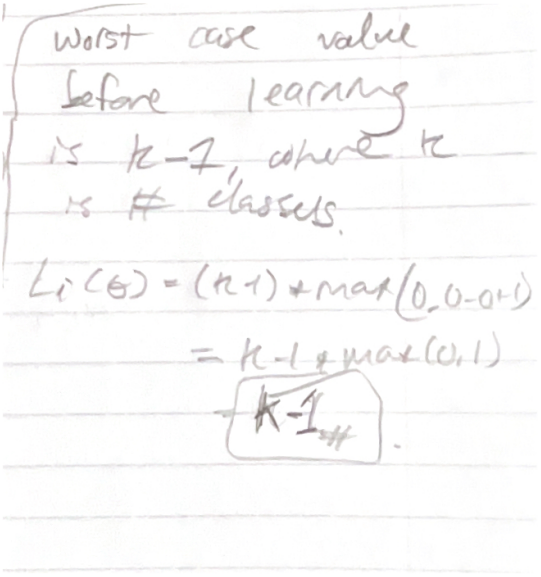


Ultimately the idea is to introduce a margin in order to train the discriminant to separate the classes as best as possible and that the classes are separated as far as possible.
Multi-class hinge loss:

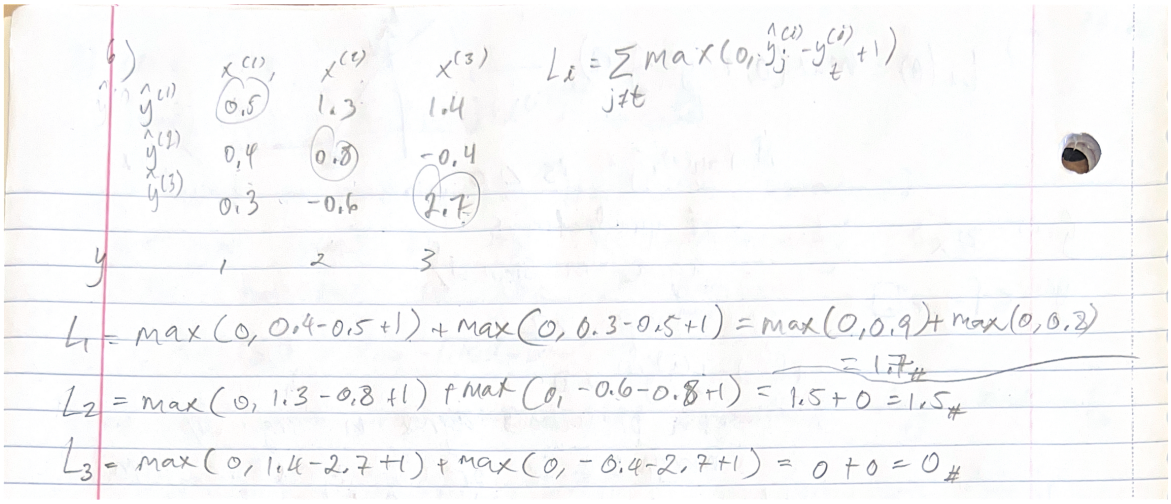


Squared hinge loss & worst case expected value:



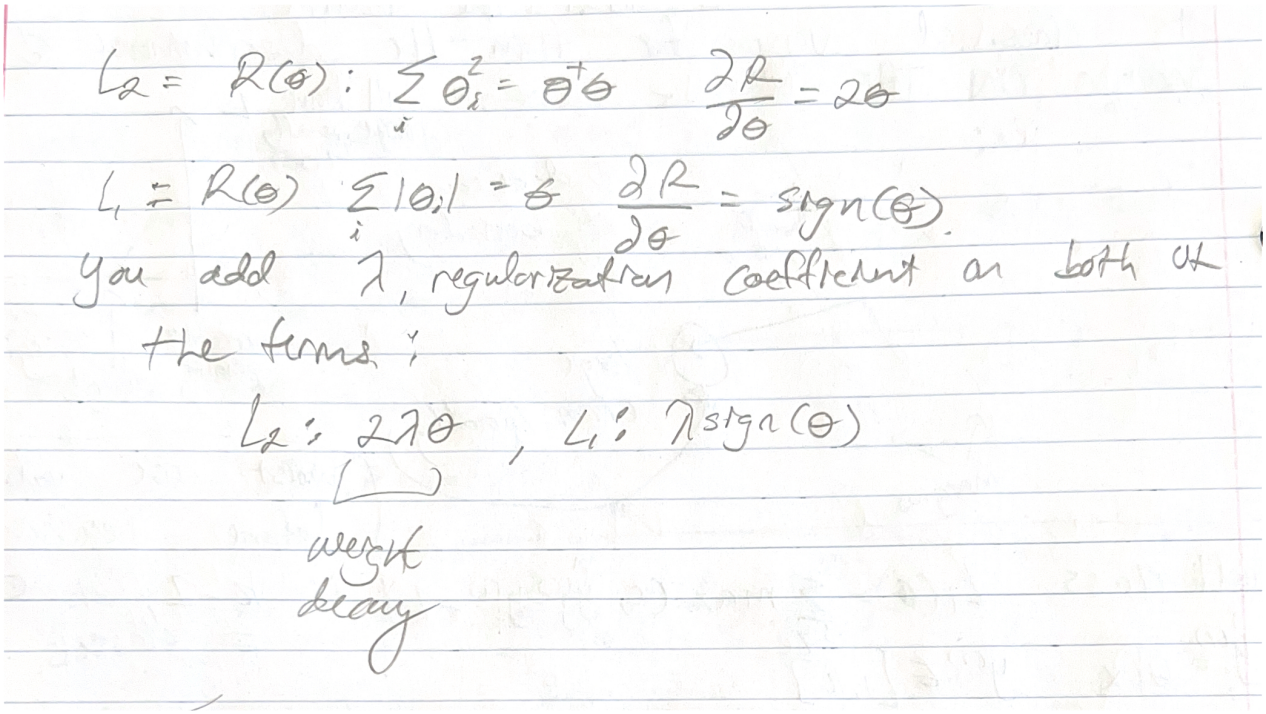


6)



7) Adding a regularization term can tackle overfitting. The purpose of the regularization term is to introduce an additional penalty to the loss term. By doing this it increases generalizability (more bias). The primary difference between the two are that L1 regularization makes the weights more sparse whereas L2 regularization spreads the weights. L1 reg. is just the absolute value of the weights matrix whereas L2 reg. is the squared value. The way to choose is a regularization coefficient is simply by trial and error (i.e. hyper parameter tuning). You can just use a scheme such as grid-search or something to find the optimal regularization term by trial and error (pick a term, train the model, evaluate the performance, change the term and repeat and choose the one that results in the best performance).

8) The L1 and L2 loss have additional terms when dealing with gradient descent. For L1 it's λsign(θ) and L2 it's 2λθ. The regularization terms change the penalty based on the weights. The way they actually change the loss is mentioned up above in my answer to question 7, and here is the derivation:



9) Kernel Regularization:
Regularization of the parameters that reduce the weight and introduce a weight decay on the weights 'w'.

Bias Regularization:
Regularization of the bias terms, aims to reduce the bias and expects smaller output values.

Activity Regularization:
Regularization of the parameter y_hat by introducing a decay on the output y_hat and reducing the output value, bringing it closer to lower values.

Optimization

1. The advantage of using back-prop to compute gradients numerically is that they're easier to compute and use symbolic derivatives. The numerical approach may also in some cases be slower. One possible use of the direct computation of gradients is perhaps for verification purposes.

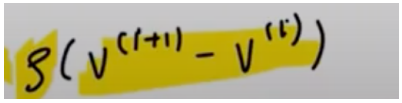
2. Stochastic gradient descent is when you update the weights with batches from the training dataset. Whereas in normal gradient descent you update the weights after every epoch (i.e. seeing the entire training dataset). SGD is faster because you're updating the weights sooner as you progressively see the batches.

3. Larger batch sizes serve to decrease the variance for the SGD updates but may end up deterring the model performance the same way a model can over-fit to the data. Smaller batch sizes may take longer to converge and will simply approach the GD algorithm but would also improve the bias of the model. The four main problems with SGD are as follows:

(1) What should be the optimal LR?
(2) What happens if the loss is too sensitive to our parameters, i.e. updates too fast in one direction or too slow in another.
(3) How can you avoid getting stuck in a local minima / saddle point
(4) Mini-batch gradient estimates may be noisy

4. SGD w/ momentum takes care of poor conditioning by smoothing it out by averaging the gradient previous gradients.
It addresses saddle points because there is a velocity vector that is also calculated at each step, so if it reaches a local minimum (where the gradient may be 0) the idea is that the velocity vector will still be an element that will remain and will help to continue the iteration and move us out of the local minimum.
And lastly, it addresses noisy gradients by smoothing them out via moving average.

5. NAG is more accurate than simple momentum in the sense that it provides a more precise update for the gradient because it's calculated at the future time-step. It's called accelerated because of the following term in the update equation:

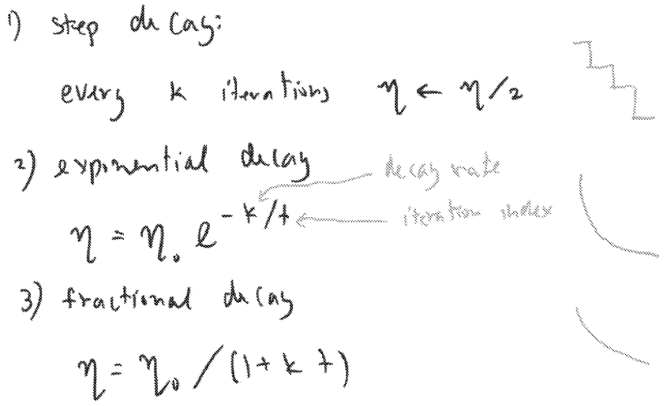


Difference of two velocities --> acceleration.

6. There is a step decay where by a certain step you simply may choose to half the learning rate (for example).

There is exponential decay, where the learning rate starts off high and exponentially decays. You may tune the decay rate and the iteration index of when the decay will occur.

And lastly there is fractional decay whereby the learning rate decays by some given fraction. Similar to the exponential decay case, you may tune the decay rate and iteration index of when the decay will occur.

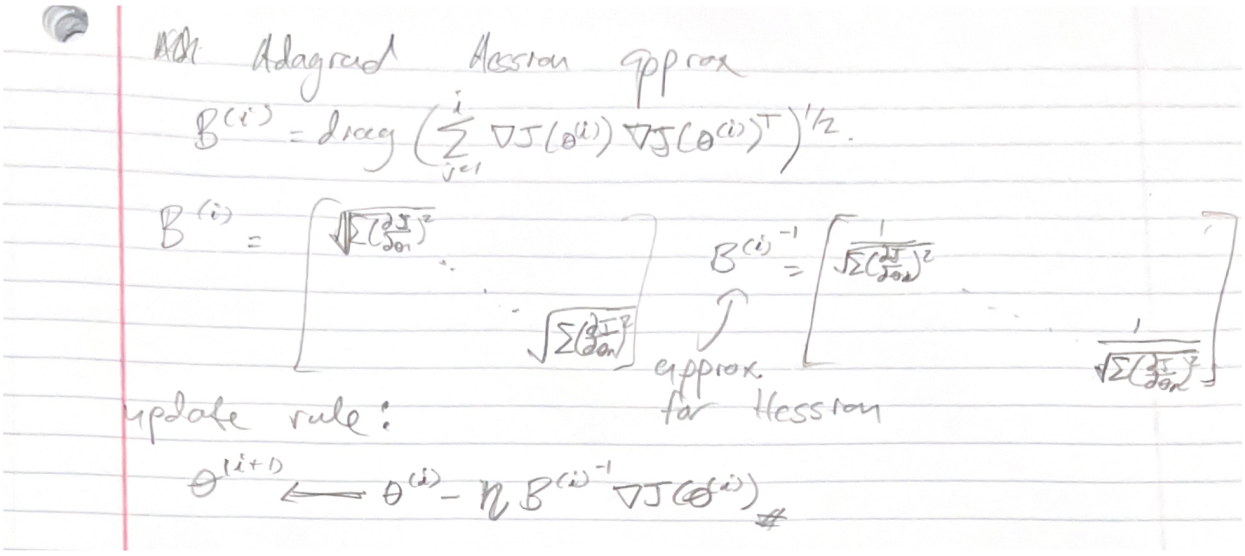


7. The learning rate in the newton's method is computed by taking the inverse of the hessian matrix which is the second order partial derivative matrix with respect to components in the theta vector. The basic meaning behind the hessian matrix is that it is a good estimate of the sensitivity of the LR with respect to each of the different parameters. It helps to resolve issue number **2** (Loss too sensitive to parameters) from problem (3).

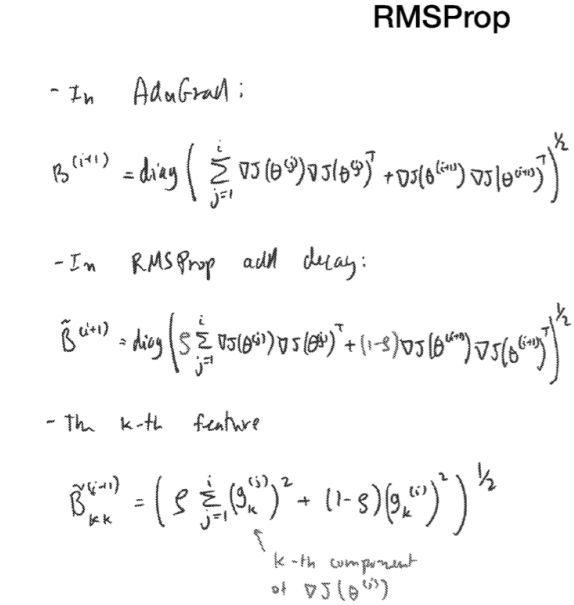
8. The condition number helps to define the sensitivity using the singular value of the Hessian matrix:
condition number = largest_singular value / smallest_singular value

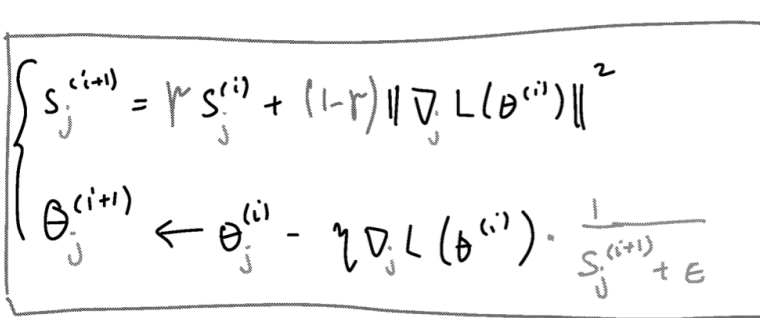
The problem gets more difficult when the condition number is high because that means that the matrix is more sensitive to error in the input which ultimately may lead to improper optimization. This is often the case when there is poor conditioning.

9. Adagrad approximates the inverse of the hessian matrix by computing the matrix B(i)^-1:



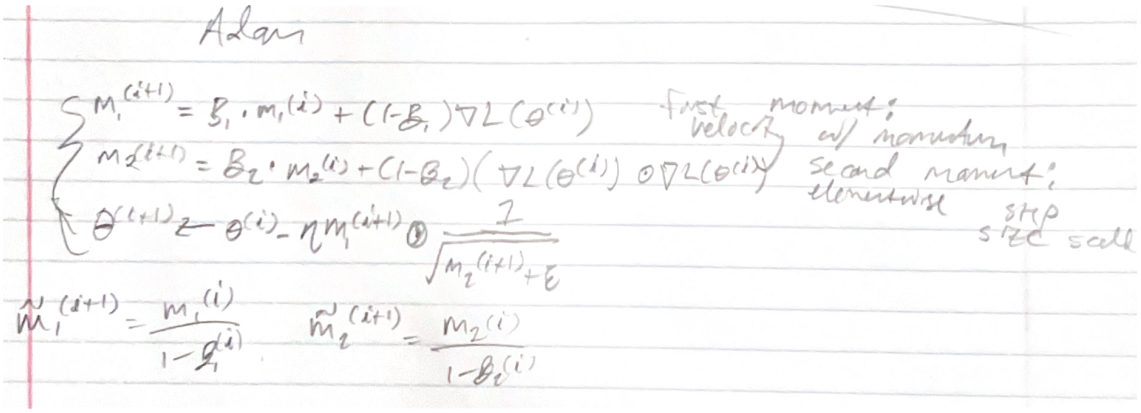
10. In Adagrad since we normalize by element-wise sum of the square gradients, the step size will become smaller as the iterations progress. To combat this, RMSprop introduces a decay factor when adding new gradients to the gradient sum.



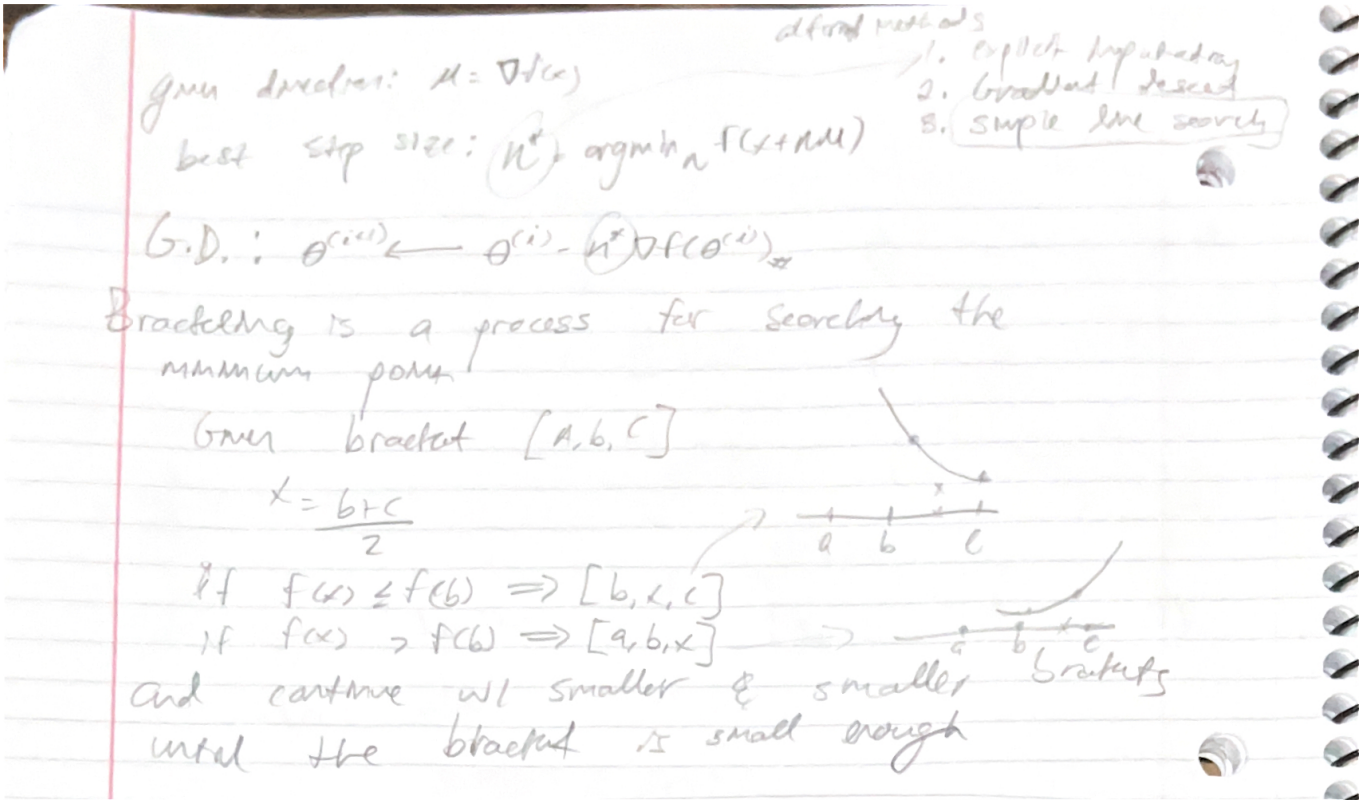


11. You use a bias correction term because in the beginning when m1 = m2 = 0 will result in too large of a step from the second moment in the update equation. So to prevent the m2 from being 0 we add a bias correction term that divides the number based on the iteration number so as to increase the initial moment and therefore prevent the update from being too large since the moment we are interested in is in the denominator and a larger m2 will serve to decrease the overall update:

Also, Adam combines RMSprop and momentum by incorporating velocity w/ momentum (first moment) and element-wise step scaling (second moment) to the update equation:



12. The idea behind line search is that instead of a fixed step size try and find the 'optimal' step size in a given direction.



An alternative to bracketing can be coordinate descent. The advantage is that it's more accurate but the disadvantage is that it's computationally more expensive.

13. Quasi-newton methods basically approximate the inverse of the hessian matrix via gradient evaluation. The advantage of BFGS algorithm over Newtons is the time complexity. In BFGS it's O(N^2) whereas in Newtons its O(N^3). The advantage of BFGS when comparing with Adam is that it's more accurate and but the disadvantage is that it's more computationally expensive than Adam.

Regularization

1. You multiply each coefficient by p, where p E [0,1] and as the iterations progress, weights that are not multiplied by the coefficient and therefore reinforced, decay to 0. This process is effectively equivalent to adding a regularization term to the loss.

2. Early stopping basically halts the training process once the model begins to overfit, i.e. when the models validation loss begins increasing and the validation accuracy begin decreasing but the training loss continues to decrease and the training accuracy continue to increase.
Strategy to reuse validation data:
1. Once early stopping kicks in and you know the number of epochs where the model begins to overfit, you retrain the model and combine the validation and training set and train only up to the number of epochs before it overfits.
2. You can continue to train from the previous weight using the entire data when the validation loss is bigger than the training loss.

3. Data augmentation is creating data with transformation etc. applied from the existing feature data set and introducing this newly created data into the feature set for model training. The goal is to improve variability, generalizability, and overall performance in the model by increasing the data at hand for the model to learn with.

4. Dropout is performed by dropping (or a better term is deactivating) random nodes (with probability of 1-p) in fully connected layers where p is a hyper-parameter. The main idea behind this is that it addresses overfitting by improving the generalizability of the model by forcing other neurons (nodes) to learn features when the various other neurons are dropped (turned off / deactivated by dropout). This helps reinforce the neurons' feature detection capability and just overall helps the model to overfit less.

The disadvantage is that it may take longer for the model to properly train since not all units are being used in the training stage (since some are being dropped due to dropout).

5. Multiplying the output of each node by P is equivalent to computing the expected valued for 2^n dropped-out networks.

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

mask for all n models

6. In training batch normalization is performed by taking the mean and standard deviation of the mini batch at the respective layer / neuron and then using the normalization equation of subtracting by the mean and dividing by the standard deviation. So that'll leave us with values created (mean and standard deviation) for all the respective times we normalize a batch, and when we get to inference we just average the values out (for the mean and standard deviation for the batches) that were created and use that mean and standard deviation.
The randomness for batch normalization gets introduced by the input batch.

7. So some saturation is actually needed for the training to stop and the purpose of batch norm is to reduce saturation but not so much so that the model never trains. The scale and shift parameters serve to introduce

some saturation that is needed for the model to converge. So when gamma_j and beta_j (the parameters used to scale and shift) are equal to standard deviation and the mean (respectively) the normalization will be cancelled. The values can be learned simply by calculating gradients with respect to the parameters (gamma_j and beta_j) and updating the parameters with optimization process. And a good initial value is to set the mean to 0 and the standard deviation to 1, so gamma_j and beta_j = 1, 0 respectively.

8. When we train multiple models that are independent and use the average to make a prediction we reduce overfitting because the prediction is now an average of multiple models which is much less sensitive to the bias and variance of a single model. There are some strategies to obtain multiple models:

1. change the data
2. change the parameters
3. you can record multiple snapshots of the model during the training phase (and vary learning r