

cs577 Assignment 3

Girish Rajani-Bathija

A20503736

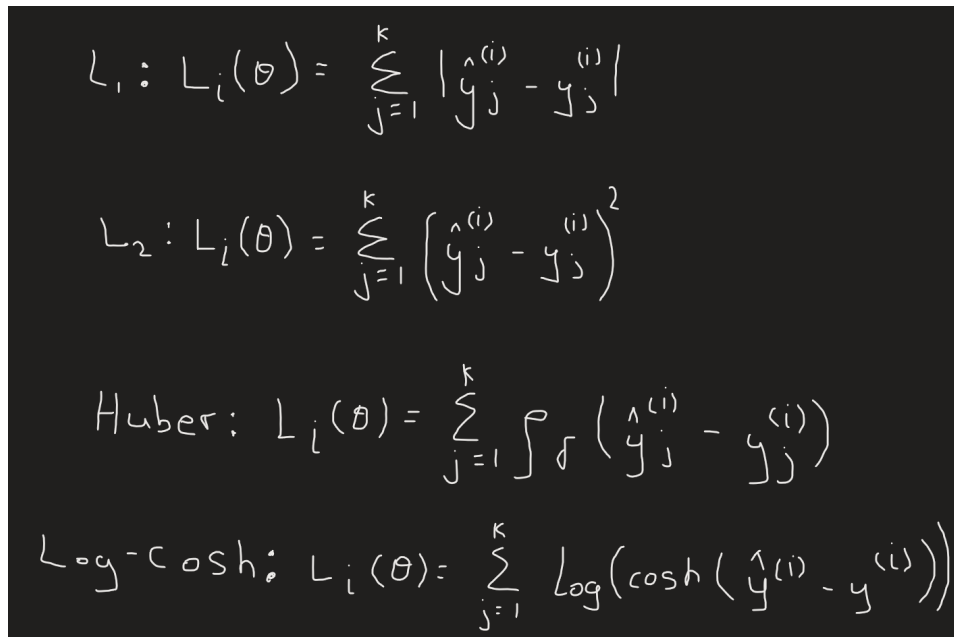
Department of Computer Science

Illinois Institute of Technology

October 25, 2022

Loss

1. Write the equations for L1, L2, Huber, and Log-cosh loss functions and compare them. Explain the advantage or purpose of each loss function.


$$\begin{aligned} L_1: L_i(\theta) &= \sum_{j=1}^k |\hat{y}_j^{(i)} - y_j^{(i)}| \\ L_2: L_i(\theta) &= \sum_{j=1}^k (\hat{y}_j^{(i)} - y_j^{(i)})^2 \\ \text{Huber: } L_i(\theta) &= \sum_{j=1}^k \int d (\hat{y}_j^{(i)} - y_j^{(i)}) \\ \text{Log-cosh: } L_i(\theta) &= \sum_{j=1}^k \log(\cosh(\hat{y}_j^{(i)} - y_j^{(i)})) \end{aligned}$$

The L2 loss is more sensitive to outliers than L1 loss, meaning that if there is a small mistake on the L2 loss, the impact is much greater than if a mistake was made on the L1 loss. This is because, as shown in the equation for L2 loss, the penalty is squared (d^2 compared to $|d|$ in L1 loss), which results in a larger penalty if a mistake is made. L2 when compared to L1 is easier to solve since we can get a more stable solution while with L1, it is a bit difficult since the derivative is not continuous.

The advantage of Huber loss is that if the value of 'd' is small, you get a quadratic error (L2 loss), and if the value of 'd' is large, then you get a linear error which means that the influence of the 'd' is capped, so the impact will not be so great if a mistake was made. This method of huber loss essentially solves the major problems of L1 and L2 losses. Log cosh loss is somewhat similar to L2 but instead, it reduces sensitivity to outliers so it is better than L2 loss in the sense that if there is a small mistake, the impact will not be as great as it would be in L2 loss. Logcosh loss is differentiable everywhere while Huber loss is only differentiable once.

- Write the equations for cross-entropy loss and explain how it is derived using maximum log-likelihood. Explain the worst cross-entropy loss value you expect for random assignment.

$$L_i(\theta) = - \sum_{j=1}^K y_j^{(i)} \log(\hat{y}_j^{(i)})$$

To derive the equation for cross-entropy loss using maximum log-likelihood, we first use the maximum likelihood function to find the negative log-likelihood function. This is then used as the loss function for cross-entropy loss. This can be illustrated below.

$$\begin{aligned} \text{Maximum Likelihood: } L(\theta) &= \prod_{i=1}^m \prod_{j=1}^K (P(y=j | x^{(i)}))^{y_j^{(i)}} \\ -\log \text{ likelihood: } \ell(\theta) &= -\log L(\theta) \\ &= - \sum_{i=1}^m \sum_{j=1}^K y_j^{(i)} \log(P(y=j | x^{(i)})) \\ &= - \sum_{i=1}^m \sum_{j=1}^K y_j^{(i)} \log(\hat{y}_j^{(i)}) \end{aligned}$$

The worst cross-entropy loss value you can expect for random assignment is $\log(k)$. This is a bad loss value because it means that the classifier will return probabilities of $1/k$ for all of the outputs as it will not know what they are and so it just distributes the same probability.

- Write the equation for softmax loss and describe when to use it.

$$\ell(\theta) = - \sum_{i=1}^m \sum_{j=1}^K y_j^{(i)} \log(\hat{y}_j^{(i)})$$

For the softmax loss, the softmax activation is only used in the final layer of a k -class classification and uses cross-entropy for loss. Softmax loss is the first choice for multi-class classification as it produces a probability distribution where the sum of all classes must add up to 1, and the class with the highest probability would be where the index is classified.

4. Write the equation for Kullback-Liebler loss and explain its meaning. Explain the circumstances under which there is no difference between using cross-entropy or Kullback-Leibler to train the network.

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

The Kullback-Liebler divergent measures similarity between distributions so if we assume to have 2 distributions, $p(x)$, and $q(x)$ then the similarity of p and q are compared. We then define the likelihood ratio for the entire dataset and it will be a product of p/q . The log-likelihood ratio can then be derived from this to find if $p(x)$ or $q(x)$ is the better model. When comparing this to the equation for Kullback-Liebler loss, we can think of $p(x)$ as the true value and $q(x)$ as the prediction. Using the log-likelihood ratio, we compute the expectation value to get the KL divergent, which is the expected value of the log of the likelihood ratio of the two distributions p and q and as we can see in the KL loss equation, it means the same thing.

The Kullback-Liebler loss can be broken down into the $\log(y(i)_j) - \log(\hat{y}(i)_j)$, and what this means is that the left side is the -entropy and the right side is the same as the cross-entropy so when the sum of the negative entropy is always the same and does not change then there is no difference between using cross-entropy or Kullback-Leibler to train the network

5. Explain Hinge loss and squared Hinge loss. Describe the fundamental idea behind it and the worst value you expect for it before learning.

The hinge loss comes from an SVM loss. When looking at a linear discriminant, we learned that there is 1 decision boundary to classify a 2-class problem, and now we are introducing 2 margins, 1 on either side of the decision boundary. This means that the examples not only need to be on the right side of the decision boundary but should also be outside of the margin. When looking at hinge loss, if the example is on the correct side of the margin then there is no penalty/no contribution to the loss. If the example is inside the margin but still on the right side of the decision boundary then there is some contribution to the loss but if the example is on the incorrect side of the boundary then the contribution to the loss increases more. Therefore, with hinge loss, it increases with distance. Squared hinge loss simply takes the hinge loss function and squares it. This is done to get a smoother loss penalty when the distance from the boundary increases compared to hinge loss.

The worst value expected from hinge loss before learning is infinity. This occurs when the example is on the wrong side and very very far from the classifier then it significantly contributes to the hinge loss.

6. Hinge Loss:

	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
\hat{y}_1	0.5	0.4	0.3
\hat{y}_2	1.3	0.8	-0.6
\hat{y}_3	1.4	-0.4	2.7
y	1	2	3

$L_1(\theta) = \sum_{j=1}^n \max(0, y_j^{(1)} - \hat{y}_j^{(1)} + 1)$
 $L_1 = \max(0, 1.3 - 0.5 + 1) + \max(0, 1.4 - 0.5 + 1)$
 $= 1.8 + 1.9 = 3.7$

$L_2 = \max(0, 0.4 - 0.8 + 1) + \max(0, -0.4 - 0.8 + 1)$
 $= 0.6 - 0.2 = 0.4$

$L_3 = \max(0, 0.3 - 2.7 + 1) + \max(0, -0.6 - 2.7 + 1)$
 $= 0 + 0 = 0$

7. Explain the purpose of adding a regularization term to the loss function. Explain the difference between L1 and L2 regularization and how they affect the weight distribution in the network. Explain the way to choose the regularization term coefficient.

By adding a regularization term to the loss function, we can modify the loss term to include a preference for smaller models and this is preferred because smaller models are simpler hence producing a more stable solution that will generalize better and be less likely to overfit. This is done by making the weights of the network small.

L1 Regularization:

$$R(\theta) = \sum_{i,j} |\theta_{i,j}|$$

L2 Regularization:

$$R(\theta) = \sum_{i,j} \theta_{i,j}^2$$

With L1 regularization, we have the sum of all the absolute values of all the coefficients but with L2 regularization, we take the sum of the square of the coefficients. Although they appear to be similar since they both drive theta (weights) down towards zero but as it relates to the weight distribution, L1 regularization makes weights sparse, meaning that it concentrates the weights more, whereas the L2 regularization makes the weights smaller, while spreading them more. To choose the regularization term coefficient, we can use GridSearch on all values we can think of, or use random values iteratively.

8. Explain how L1 and L2 loss terms affect gradients in the network.

Normally, in the gradient update, we subtract the learning rate * the gradient of the loss but now that regularization is introduced, we also subtract the learning rate * the gradient of the regularization. The 2 terms can be seen below.

$$\frac{dL}{d\theta} = \frac{d}{d\theta} \frac{1}{n} \sum_{i=1}^n L_i(f(x^{(i)}, \theta), y^{(i)}) + \lambda \frac{d}{d\theta} R(\theta)$$

L1 loss terms affect the gradients in the network because during the gradient descent when using the L2 loss term, we subtract $\lambda \text{sign}(\theta)$ in the update. As shown above, underlined in red is the regularization term so when using L1 loss, we will subtract $\lambda \text{sign}(\theta)$. This will take the sign of the θ so, whether that is +1 or -1, and multiply it by λ .

L2 loss term affects the gradients in the network because during the gradient descent when using the L2 loss term, we subtract $\lambda \theta$ in the update. As shown above, underlined in red is the regularization term so when using L2 loss, we will subtract $\lambda \theta$. This results in weight decay as the θ is being subtracted more and more and will be decayed towards 0 if it is not being reinforced.

9. Explain the difference between kernel, bias, and activity regularization.

When working in Keras, the artificial neuron is derived by $\hat{y} = r(wx+b)$ where w is the kernel, b is the bias and \hat{y} is the activity. In regularization in Keras, we can specify to regularize w , b , or \hat{y} which simply means adding some decay to decay them to 0.

Normally, we perform regularization on the kernel which means performing weight decay to lower the coefficients θ . Sometimes, bias regularization is done when we expect small values in the output. This drives the values in the output of the units to lower values. Sometimes, activity regularization is done when we expect the function to output values close to zero.

Optimization

1. Explain the advantage of backpropagation over direct numerical computation of gradients (using the definition of partial derivatives). What is a possible use of direct numerical computation of gradients.

The advantage of backpropagation over direct numerical computation of gradients is that it is very expensive to perform direct numerical computation. This can be further explained by using the definition of partial derivatives:

$$\frac{\partial f}{\partial x_i}(x) = \lim_{h \rightarrow 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_n)}{h}$$

Where $f(x_1, \dots, x_n)$ is the vector where I am currently at

The derivative with respect to each of the parameters of x is computed numerically. As mentioned earlier, it is expensive to compute the above equation because we need to push all the batch through the network to get 1 value then push another set of values (without the h), and then take the difference between the two. What this does is gives 1 derivative with respect to 1 parameter so if there are many parameters such as a million parameters then this can get quite expensive. It is easy to compute but is slow. Backpropagation uses the chain simple derivatives. It is easy to compute and uses symbolic derivatives in Python.

A possible use of direct numerical computation of gradients is it is used for verification (gradient check on simplified network).

2. Explain the difference between stochastic gradient descent (SGD) and gradient descent (GD). Which one is expected to converge faster and why?

Gradient descent:

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \gamma \nabla L(\theta^{(i)})$$

For gradient descent, we take the current value and subtract learning rate * the gradient and get a new value.

For Stochastic gradient descent:

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \gamma \nabla L_j(\theta^{(i)})$$

In SGD, we randomly order the examples, then perform a loop over all the examples ($j = 1 \dots m$). As you can see, the update rule for SGD is similar to GD but instead, we use the loss with respect to the j th example. From m examples, we take 1 example (j), compute the loss and gradient then update the parameters.

SGD is expected to converge faster because, with SGD, we get very frequent updates of the gradients because we don't have to wait for the entire dataset however, the update is less accurate than GD.

SGD updates after processing every example (faster), and GD updates after processing all examples (more accurate).

3. Explain the tradeoff in selecting the batch size for SGD. Explain the 4 main problems with SGD.

The tradeoff in selecting the batch size for SGD is the mini-batch gradient descent. Instead of iterating through the entire dataset, it processes updates for smaller batches of the training dataset

4 main problems with SGD:

- What should be the learning rate? - We know the direction but we don't know the step size. It has to be guessed.
 - What happens if the loss is more sensitive to one parameter (too fast in some directions and too slow in others) - some weights may be too sensitive or less sensitive to a learning rate so we cannot use the same learning rate throughout. This is a problem because then we will have to tune the learning rate for each parameter and if we have let's say, a million parameters then it is not possible to fine-tune each learning rate.
 - How to avoid getting stuck at local minimum or saddle points - In gradient descent, we are looking for a minimum of the loss function but we may get stuck where the gradient of the loss = 0 but we did not get to the final location so we want to know how to avoid getting stuck at this local minimum.
 - Minibatch gradient estimates are noisy - When we use minibatch instead of the entire set of examples, it is noisy and so we want to know how we can make the minibatch gradient estimate less noisy.
4. Explain how SGD with momentum addresses poor conditioning, minimum/saddle points, and noisy gradients.

SGD with momentum addresses minimum/saddle points because even at the minimum/saddle points, there is a velocity so unlike SGD where it would stop because the gradient is 0, there is still some residual momentum from before which will allow SGD with momentum to continue and then find another minimum point, therefore avoiding getting stuck at local minimum points.

SGD with momentum addresses noisy gradients because we average the gradient with the previous velocity so even if the gradient estimates are noisy because it is averaged, it will not be so noisy.

SGD with momentum addresses poor conditioning because it's smoothed out by averaging with previous gradients.

5. Explain the advantage of the Nesterov Accelerated Gradient (NAG) momentum vs simple momentum. Explain the term "accelerated" in NAG.

NAG momentum computes the gradient at future time step whereas simple momentum computes gradient at current step. This means so that the gradient would be more accurate in the future time step when computed in NAG momentum compared to simple momentum.

The term accelerated in NAG comes from the update of $\theta^{(i+1)}$. This update can be seen below.

$$\theta^{(i+1)} \leftarrow \theta^{(i)} + v^{(i+1)} + \rho(v^{(i+1)} - v^{(i)})$$

As shown, underlined in green above, that is the difference between 2 velocities which reminds us of acceleration and this is how NAG gets the term “accelerated”.

6. Explain possible strategies for learning rate decay.

Strategies for learning rate decay:

1. Step decay:

With every k -iterations, we half the learning rate so after every iteration, we will see the learning rate getting smaller and smaller. With this strategy, it abruptly changes the learning rate.

$$\eta \leftarrow \eta / 2$$

2. Exponential decay:

This strategy gradually changes the learning rate. The base learning rate (η_0) is multiplied by the exponent as shown below which results in a gradual decay.

$$\eta = \eta_0 e^{-k/t}$$

Where k is the decay rate and t is the iteration index

3. Fractional decay:

This strategy also starts with a base learning rate but this time it divides by $(1+kt)$ so as t gets bigger and bigger, we are dividing by bigger and bigger numbers which results in the learning rate getting smaller. This can be seen below:

$$\eta = \eta_0 / (1 + kt)$$

7. Explain how the learning rate is computed using Newton's method and explain the meaning of the Hessian matrix.

Newton's method computes the learning rate by starting with a guess X_0 and then finding the update ΔX such that $f(X + \Delta X) = 0$. This can be seen below.

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

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

$$\nabla J(\theta_0) + \underbrace{\nabla(\nabla J(\theta_0))}_{H} \Delta \theta = 0$$

H : Hessian Matrix

The hessian matrix that was computed is the second order derivative matrix H :

$$H = \begin{bmatrix} \frac{\partial^2 J}{\partial \theta_1 \partial \theta_1} & \dots & \frac{\partial^2 J}{\partial \theta_1 \partial \theta_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 J}{\partial \theta_n \partial \theta_1} & \dots & \frac{\partial^2 J}{\partial \theta_n \partial \theta_n} \end{bmatrix}$$

Where $\theta = [\theta(1) \dots \theta(n)]$

Using the equation computed from earlier, we can compute the update rule using the Hessian matrix:

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

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

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

8. Explain the condition number and what happens when there is poor conditioning.

The condition number computes the singular value of the Hessian matrix and takes the ratio of singular values of A with respect to its smallest singular value. When there are high-condition numbers, the problem is more difficult to solve. The condition number can be shown below:

$$\frac{sv_1}{sv_n} \leftarrow \text{largest sv}$$

$$\frac{sv_1}{sv_n} \leftarrow \text{smallest sv}$$

There is an advantage of using Newton's method when we have a problem that is not conditioned well. Poor conditioning means there are some parameters where the change in the direction is more compared to others.

9. Explain the way in which the AdaGrad algorithm approximates the inverse of the Hessian.

AdaGrad approximates the inverse of the Hessian matrix without actually computing the Hessian matrix to avoid storing and inverting it. This approximation is done by investing of computing the Hessian matrix, H, we compute a matrix, B which is iterative. This can be illustrated below:

$$B^{(i)} = \text{diag} \left(\sum_{j=1}^n \nabla J(\theta^{(i)}) \nabla J(\theta^{(i)})^T \right)^{1/2}$$

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

When looking at how the scale factors are computed, we can see that we add up the sum of squares of the derivatives with respect to that parameter. For example, when we want to compute the update for $\theta(1)$, we use a scale factor that sums the derivatives of the loss function with respect to $\theta(1)$, then squares it, and square root's it. In one scenario where the loss J is sensitive to the parameter $\theta(1)$ then, this means that a small change in $\theta(1)$ will cause a big change in the loss (the derivative of J with respect to $\theta(1)$ will be high). When J is sensitive to $\theta(1)$ then this will cause the scale factor in the inverse approximation to be low which means that the update will get a low weight resulting in updates not very fast which is preferred.

As seen above, the matrix B is diagonal, unlike the Hessian matrix. The matrix contains the scale factors that will be applied to the different coordinates. After applying the scale factors, we get the inverse approximation for matrix B which can be seen below.

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

10. Explain the problem with AdaGrad algorithm and how RMSProp addresses it.

The problem with AdaGrad algorithm is that because we normalize by elementwise sum of square gradients, the step size will become smaller as iterations progress. We want to control this because we do not want it to go too fast. To control this using RMSProp, we use a decay factor/weight (e.g 0.9) when adding new gradients to the gradient sum. This can be seen below:

In Adagrad:

$$B^{(i+1)} = \text{diag} \left(\sum_{j=1}^i \nabla J(\theta^j) \nabla J(\theta^j)^T + \nabla J(\theta^{(i+1)}) \nabla J(\theta^{(i+1)})^T \right)^{1/2}$$

In RMSPROP add decay:

$$B^{(i+1)} = \text{diag} \left(\beta \sum_{j=1}^i \nabla J(\theta^j) \nabla J(\theta^j)^T + (1-\beta) \nabla J(\theta^{(i+1)}) \nabla J(\theta^{(i+1)})^T \right)^{1/2}$$

We can see that instead of just adding the gradients in Adagrad, RMSProp we add the gradient with a weight and give a weight to the previous sum which will prevent the matrix B from getting bigger and bigger.

11. Explain how the Adam algorithm combines RMSProp with momentum. Explain the need for bias corrected term.

Adam algorithm combines RMSProp (scale by sum of gradient elements) with momentum. In the Adam algorithm, there are 2 moments that are computed (m1 and m2) which are updated iteratively. The computation for the moments and Adam update can be seen below.

$$\begin{aligned}
 m_1^{(i+1)} &= \beta_1 \cdot m_1^{(i)} + (1 - \beta_1) \Delta L(\theta^{(i)}) && \text{first momentum:} \\
 &&& \text{velocity with momentum} \\
 m_2^{(i+1)} &= \beta_2 \cdot m_2^{(i)} + (1 - \beta_2) (\Delta L(\theta^{(i)}) \odot \nabla L(\theta^{(i)})) && \text{second momentum:} \\
 &&& \text{Elementwise step size scale} \\
 \theta^{(i+1)} &= \theta^{(i)} - \frac{1}{\gamma} \frac{m_1^{(i+1)}}{\sqrt{m_2^{(i+1)} + \epsilon}}
 \end{aligned}$$

The first moment looks like the momentum and the second moment looks like the RMSProp so then the update for Adam to get the new value of theta, we take the previous value of theta - learning rate * m1 (the direction where we want to take the step) * 1/(m2+epsilon) (which is the adaptive scaling along different directions that it depends on the history of the gradients).

Since the moments are initialized to 0 in the beginning, in the update rule when dividing by the second momentum we will get a large step because we then divide by Epsilon and since it is a small number, we get a very large number. To deal with this, we use a bias correction term which is done by dividing the momentum by a number depending on iteration number (so that the initial moments are larger). This means that the bias correction term of m2, m2 tilde, will not be as small as if we use m2 directly and this will mean that when we compute the Adam algorithm, we will get a smaller step.

12. Explain gradient descent with line search and the bracketing algorithm. Describe an alternative to bracketing and the advantages/disadvantages of bracketing compared with this alternative.

In gradient descent, we knew the direction where to go but not the step size so with gradient descent with line search, instead of a fixed size, we find the “best” step size. Given the search direction, we find the best size by computing the following:

$$\gamma^* = \underset{\gamma}{\operatorname{argmin}} f(x + \gamma u)$$

The update is given as follows:

$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \gamma^* \nabla f(\theta^{(i)})$$

Where instead of using a standard learning rate, we use the optimal step computed multiplied by the direction gradient.

Bracketing search is a cheaper way to solve for the optimal step. We start with a bracket $[a,b,c]$ and want to find a smaller bracket where the minimum point is. So we essentially take the bracket and make it smaller until we can find where the minimum point is. We compute a middle point between b and c which is x computed as $x = (b+c)/2$ and we can conclude that if $f(x) \leq f(b)$ then the next bracket will be $[b,x,c]$ and if $f(x) > f(b)$ then the next bracket will be $[a,b,x]$. We continue this process iteratively and get a smaller and smaller bracket. We do this until the bracket is small enough. We may not get the smallest value but will get a small enough bracket which is cheaper.

An alternative to bracketing is coordinate descent, where we start with a $\theta(0)$ and direction set $\{u^{(i)}\}$. We take steps along the coordinate and go in horizontal and vertical lines without computing the optimal direction so our $\{u^{(i)}\}$ will be $\{e(i)\}$, which is something other than gradients.

13. Explain quasi-newton methods. What is the advantage of the BFGS algorithm over Newton. What are the advantages/disadvantages of BFGS compared with Adam.

Inverting the Hessian matrix is expensive because since Hessian has n^2 elements, inverting it will have $O(n^3)$ time complexity. Instead, the quasi-newton methods (BFGS or limited memory BFGS) approximate the inverse of the Hessian matrix using gradient evaluations. It requires a large number of examples. This is done by the following steps:

1. Compute the quasi-newton direction

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

2. Determine step size (eg. optimal step learning rate through bracketing line search)
3. Compute parameter update:

$$\theta^{(i)} = \theta^{(i-1)} + \gamma^* \Delta \theta$$

4. Compute updated Hessian approximate $H^{(k)}$

The advantage of BFGS over Newton is that it is a less expensive computation in that we approximate the inverse of the Hessian matrix instead of actually computing it fully. This means that now the time complexity is reduced from $O(n^3)$ to $O(n^2)$.

Regularization

1. Explain how weight decay is related to adding a regularization term to the loss function.

In weight decay, we multiply each coefficient by $P \in [0,1]$ and as the iteration progresses, weights that are not reinforced get decayed to 0. This method is equivalent to adding a regularization term to the loss function.

2. Explain how early stopping to prevent overfitting is performed. Explain the strategies to reuse the validation data.

Instead of training until we get the minimal loss, instead we stop before it is minimal because we believe that at some point we will overfit so to prevent this from happening, we perform early stopping. This is done by looking at the validation loss, once we see that the validation loss starts going up then we know that we are overfitting and will perform an early stop at that point to prevent overfitting. Essentially, we are going to stop when the validation error increases instead of stopping when the training error stops decreasing.

Strategies to reuse the validation data:

1. Strategy #1 - Retrain on all data using the number of iterations determined from validation - This means that we take our x_{train} and y_{train} and split them into $(x_{\text{subtrain}}, x_{\text{valid}})$ and $(y_{\text{subtrain}}, y_{\text{valid}})$, choose a random θ and run early stopping using the subtrain and validation sets created. This will return the optimal number of steps. After which, we simply set θ to another random value and retrain on all the training data (subtrain and valid combined) using the optimal number of steps. We just use the validation data to tell us when to stop.
 2. Strategy #2 - Continue training from previous weights with entire data while validation loss is bigger than training loss. - Instead of wanting to know the number of iterations where we start to overfit and stopping at that point, with this strategy, we want to know what objective value we start to overfit, then continue training until that value is reached.
3. Explain how data augmentation is performed and how it assists in preventing overfitting.

With data augmentation, we augment the data by adding more data to it. So, we add synthetic data to increase variability in training which helps increase generalization. This will make the model more robust to changes in the input. We can perform data augmentation in feature or data domain. We can augment by interpolating between examples or by adding noise.

Data augmentation will perform better on new unseen data since we are adding more synthetic data to it in training. This, therefore, increases generalization, which in turn, prevents overfitting.

4. Explain how dropout is performed. What are the advantages/disadvantages of dropout?

Dropout means at each training stage, we drop out some units (they will now output 0) in fully connected layers with probability of $(1-P)$, where P is hyperparameters. For example, we can say that now 50% of the units will not function, and this helps with generalization. These removed nodes, in the next iteration, are reinstated with their original weights.

Advantages:

Reduce node interactions (co-adaptation), reduce overfitting, increase training speed, reduce dependency on a single node, and distribute features across a single node.

Disadvantage:

Longer training due to dropout (not all units are available at each step)

5. Explain how the expected value of all combinations of dropped out networks can be approximated efficiently during testing.

During inference (testing), we multiply the output of each value by P (the probability to keep a node) and it is equivalent to computing expected value for 2^n (all possible combinations of dropouts) dropped-out networks.

6. Explain how batch normalization is performed during training and during testing. In what way does batch normalization introduce randomness into training?

With batch normalization, instead of just normalizing the inputs, we also normalize the outputs of the hidden layers. During training, when we are feeding the batch of examples and compute the output, we normalize the batch. For example, we take batch 1 and compute the output z values which will result in getting a μ and a σ for that batch. This is repeated for each batch and we get different normalization. During training, we compute the normalization but during testing, we average it. So, during training, all the parameters we compute such as the μ , σ , γ and the β and then during inference we use the learned γ and β and we use the average for the σ and the μ .

During training, because batches are random, batch normalization adds randomness into the training and so reduces overfitting.

7. Explain the purpose of scale and shift parameters in batch normalization. What are the values of scale and shift parameters that will cause the normalization to be canceled? Explain how the scale and shift parameters can be learned and what is a good initial value for them.

To properly perform batch normalization and avoid being too efficient/effective and to introduce saturation, we scale and shift the parameters which are performed in 2 steps. The first step is to take the input, normalize it, and take the output and scale it by γ and shift it by β .

If there is no need for batch normalization, the network can learn to cancel the normalization by introducing Sigma and Mu

8. Explain how ensemble classifiers can assist with overfitting. Describe are possible strategies for producing ensemble classifiers.

In ensemble classifiers, we train multiple independent models and use majority vote or average during testing. So if we have a regression problem, each independent model will output a value and we take the average of these values and output a final value or with classification, we have a vote of which class each model is predicting and then we take the majority vote. This helps to reduce overfitting because since each individual model can overfit and are independent, the prediction overfits in the same way, so then when we average out their decision or do majority vote then we reduce the overfitting.

Producing ensemble classifiers:

1. Ensemble produced using different training sets. With this method, we have our original dataset and build classifiers by splitting the data. This allows the classifiers to learn by identifying features that can allow it to make distinctions and classify it better but each ensemble component is weak producing inaccurate rules.
2. Ensemble produced by dropout. When building ensemble classifiers, we randomly dropout units to get different versions. We then make predictions on each version in the ensemble classifier and then average the prediction. The ensemble this way is approximated by dropout weight instead of actual averaging.