HW3 - Tuan Tran

Loss:

- L1: $L = \cancel{|\hat{y}-y|}\ \Sigma |\hat{y} - y|$
  - Used mainly in regression problems
  - Minimize the distance ~~bet~~ /error between known and predicted values where the error function is sum of absolute differences between true and predicted ~~value~~ values.
  - This loss is ~~easier~~ easy to comprehend ~~the~~ its values, and is not ~~apfece~~ affected by outliers.

    ↳ More robust

- L2: $L = \Sigma(\hat{y} - y)^2$
  - Used mainly in regression problems
  - Minimize distance /~~error~~ between known and predicted values where error function is sum of squared differences
  - This loss provides a more stable solution than L1 and generally performs better. However, this loss is more sensitive to outliers than L1.
  - ~~The lo~~ Using this loss is easier to solve compared to L1 due to its nice gradient behavior

- Huber loss:

$$\rho_\delta(d) = \begin{cases} \frac{1}{2}d^2 & \text{if } |d| \le \delta \quad \cancel{\to \text{qua}} \\ \delta\left(d - \frac{1}{2}\delta\right) & \text{otherwise} \end{cases}$$

→ Quadratic for small d
linear for large d.

$$L = \Sigma \rho_\delta(\hat{y} - y)$$

- Huber loss basically combines the good properties of L1 and L2 loss. That is, assuming ~~δ is set~~ a suitable δ is set, then Huber is more robust to outliers than L2 loss. ~~It's also~~ Huber loss also has a nice gradient property like L2 loss, therefore, ~~it~~ it's easier to solve ~~than~~ and provides a better, more stable solution than L1 loss.
- One problem is ~~we~~ we ~~me~~ need to find the "right" δ.

② Log-cosh:

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

$$L = \sum \log(\cosh(\hat{y} - y)).$$

- Log-cosh works mostly like MSE but it's not as strongly affected by outliers.
- It has all the advantages of Huber loss, it doesn't require setting δ, and it's twice differentiable everywhere.

2,

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

$$\rightarrow \cancel{\ell(\theta) =} \log(L(\theta)) = \cancel{\log} \sum_{i=1}^{m} \sum_{j=1}^{k} 1(y^{(i)}=j) \log(P(y^{(i)}=j \mid x^{(i)}))$$

$$\rightarrow \ell(\theta) = -\log(L(\theta)) = -\sum_{i=1}^{m} \sum_{j=1}^{k} 1(y^{(i)}=j) \log(P(y^{(i)}=j \mid x^{(i)}))$$

since $\hat{y}_j^{(i)} = p(y=j \mid x^{(i)}) = h_{\theta_j}(x^{(i)})$

$$\rightarrow \boxed{L = \ell(\theta) = -\sum_{i=1}^{m}\sum_{j=1}^{K} 1(y_j^{(i)}=j)\log(h_{\theta_j}(x^{(i)})) = \text{cross entropy loss}}$$

- Random class assignment $\rightarrow p(y=j \mid x^{(i)}) = \frac{1}{K}$.

$\rightarrow$ ~~Say the loss for a single example i is~~ $\cancel{L_i = -\sum_{j=1}^{K} 1(y^{(i)}=j)\log(h_{\theta_j}(x^{(i)}))}$

The loss for a single example i is $L_i = -\sum_{j=1}^{K} 1(y^{(i)}=j)\log(h_{\theta_j}(x^{(i)}))$

Since $h_{\theta_j}(x^{(i)}) = p(y=j \mid x^{(i)}) = \frac{1}{K}$.

$\rightarrow L_i = K$ and this is a high loss value.

3,

Softmax loss is not an actual loss function, but rather

a terminology for ~~a~~ a dense NN with softmax function as activation in output layer and use cross entropy for loss

Softmax loss is first choice for multiclass classification because softmax function ~~rente~~ returns probability over K classes, which works nicely with cross entropy loss function

4,

$$\text{KL loss} = L = -\sum_{i=1}^{m}\sum_{j=1}^{k} y_j^{(i)} \log\left(\frac{y_j^{(i)}}{\hat{y}_j^{(i)}}\right)$$

- ~~KL divergence is used to measure the similarity between distributions. Similarly, KL loss is used to as a similarity measurement between our true and predicted values~~

- KL loss is used to measure the dis similarity between the distribution of true values and the distribution of predicted values.

- A nice property of KL loss is that it can be extended to a minus entropy term (w.r.t $y^{(i)}$) ~~and~~ minus ~~a~~ the cross entropy loss (between $y^{(i)}$ and $\hat{y}^{(i)}$). Thus,

usage ~~us~~ of KL loss ~~is~~ is the same as usage of cross entropy loss in the case where $y^{(i)}$ does not change

↳ If $y^{(i)}$ doesn't change throughout training, then the entropy term ~~loss~~ is a constant.

→ Minimizing KL loss is the same as minimizing cross entropy loss.

5, Assume 2-class classification (multiclass would just be an extension, ~~but~~ intuition is the same)

(*) Hinge loss:

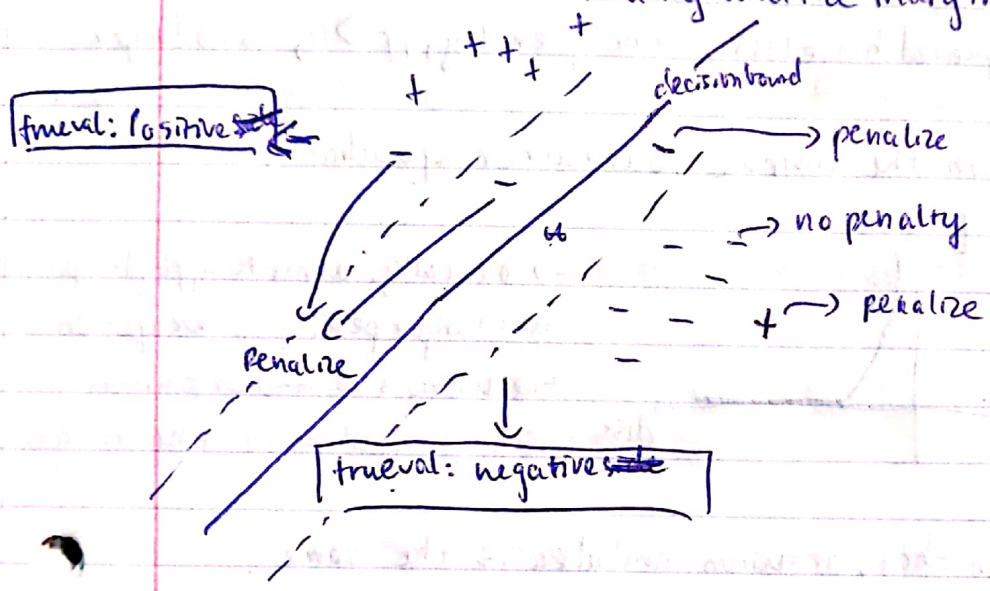
- This is a loss function used for maximum / ~~large~~ large margin classification
- The idea of large margin is that give some decision boundary, we ~~want our~~ not only want our examples to be on the right side, but we also want them to be on the ~~as far~~ right side, as far as possible from the decision boundary

↳ This acts as our confidence on how correct those examples are classified.

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

- The idea of hinge loss is that we would not penalize if the example is on the correct side ~~or~~ and on or outside the margin. However, we would ~~pena~~

penalize if example is incorrectly classified, or if example is correctly classified but it lies between the decision boundary and the margin.



- $1 - y^i \hat{y}^i$ is:
  - +, negative if predictions and true vals agree in sign ~~(same vo~~ with margin > 1.
    - $\rightarrow L = 0$ as we correctly classify and ~~the~~ it lies outside the margin. $\rightarrow$ no penalty
  - +, positive if ~~per~~ wrong predictions (predictions do not agree in sign) or correct predictions but with margin < 1.
    - $\rightarrow L = 1 - y^i \hat{y}^i \rightarrow$ penalize.

- ~~Worst value~~ Assume a k-class classification, the worst value expected before learning for a single example is $k-1$. Since this corresponds to ~~have randomly~~ having all $\hat{y}_j^{(i)}$ to be low random scores.

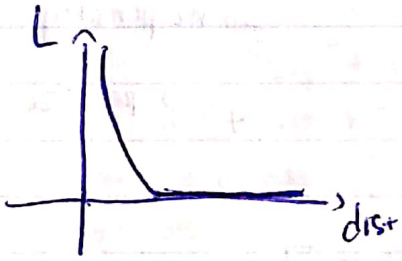
(*) Squared hinge:

$$L_i = \frac{1}{2} \max(0, 1 - \hat{y}^i y^i)^2 \quad \text{For 2-class}$$

$$L_i = \frac{1}{2} \sum_{j=1}^{k} \max(0, \hat{y}_j^i - \hat{y}_t^i + 1)^2 \quad \text{For k-class}$$

- For squared hingeloss, the penalty, if >0, is no longer linear with the distance, but rather quadratic.

↳



→ basically, wants to apply larger and larger penalty if we get on the wrong side ~~and~~ or further and further from the margin

+, Other than that, intuition and idea is the same.

~~+~~ ~~We~~ Similar to hinge loss, worst value expected : $\frac{k-1}{2}$.

6,

We have:

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

$\hat{y}_t^{(1)} = 0.5$
$\hat{y}_t^{(2)} = 0.8$
$\hat{y}_t^{(3)} = 2.7$

→ $L_1 = \max(0, 0.4 - 0.5 + 1) + \max(0, 0.3 - 0.5 + 1) = 1.7$

$L_2 = \max(0, 1.3 - 0.8 + 1) + \max(0, -0.6 - 0.8 + 1)$

$= 0.5 + 0 = 0.5$

$$L_3 = \max(0, 1.4 - 2.7 + 1) + \max(0, -0.4 - 2.7 + 1)$$

$$= 0 + 0 = 0.$$

7.

- Purpose of adding a regularization term is to decrease the magnitude of the coefficients so that we can get a simpler, more stable solution that would generalize better for new unseen data points

↳ To avoid overfitting

(*) L1 regularization:

+, Tends to make weights sparse, driving most of the weights to 0 or close to 0. That is, L1 norm produces many coefficients with 0 or ~~close to 0~~ small values and very few large coefficients

+, Sparsity of weights also lead to a feature selection property of L1 regularization since only a few selected coefficients have ~~ha~~ reasonably high values.

(*) L2 regularization:

+, Makes weights smaller while spreading them. L2 doesnot make the weights sparse, and is not as aggressive as L1 in terms of ~~driving~~ driving the weights to 0.

8,

② L2 : Since the L2 reg term is $\lambda \sum_i \theta_i^2$

and $\frac{\partial}{\partial \theta_i} \sum \theta_i^2 = 2\theta_i$

→ During ~~the~~ gradient descent, we subtract $2\lambda\theta$ when updating.

③ L1: the L1 reg term is $\lambda \sum_i |\theta_i|$

and $\frac{\partial}{\partial \theta_i} \sum |\theta_i| = \text{sign}(\theta_i)$

→ During GD, we subtract $\lambda \, \text{sign}(\theta)$ when updating

9,

Difference:

- Kernel regularization regularize the coefficients and therefore reducing coefficient values.
  → Performs weight decay.
- Bias regularization regularize the bias and therefore reducing bias values
- Activity regularization regularize both coefficients and bias; that is, it regularizes y (prediction) and therefore reducing y values.
  ↳ used when ~~the~~ our prediction is expected to have value close to 0.

Optimization:

1, - Backprop uses simple chain derivatives instead of using definition of ~~par~~ partial derivatives which is very expensive since computing a single ~~derivative~~ gradient of some function $f(\theta_1 \dots \theta_n)$ will take $\sim n$ computations

where n is the number of parameter $\theta_1 \dots \theta_n$.

↳ Backprop is easy to use and cheaper than direct numerical computation of gradients.

- Due to its accuracy ~~and~~, we can use direct numerical computation of gradients ~~to~~ for verification; that is, ~~we~~ we can check the gradients computed by some other method against direct computation.

2,

- The difference is that GD (gradient descend) only updates the parameters after processing all training instances. SGD, on the other hand, ~~only~~ updates the parameters after processing a batch of training instances. So SGD basically approximately estimates the gradients.
- SGD is expected to converge faster since it updates the parameters right after ~~pa~~ processing a random ~~sup~~ sample of data instead of having to go through all instances. This way, SGD allows rapidly computing approximate estimates of the gradient rather than slowly computing exact ~~gradients~~ gradient.

3,

- Trade off:

  +, large batch size allows SGD to be more accurate

at ~~Eat~~ gradient calculation, but will converge ~~much~~ slower due to larger number of instances.

+, Small batch size helps SGD converge faster but calculating gradients will be less accurate.

- 4 main problems with SGD:

+, We often don't know right away what should be the learning rate as this would somewhat depend on batch size. For example, ~~train~~ for a small batch size, we should have a small learning rate to maintain stability because of high variance in the estimate of gradient.

+, ~~loss maybe~~ Poor conditioning: loss is more sensitive to some parameters than ~~other~~ other parameters.
↳ More difficult to choose suitable learning rate.

+, Get stuck at local minimum or saddle points

+, The gradient estimates are noisy and can't be as accurate as GD.

4,

Adding momentum to SGD is to accelerate learning, especially in the face of high curvature, small but consistent gradients or noisy gradients.

↳ It ~~uses~~ accumulates an exponentially decaying moving average of past gradients and continues to move in their direction (velocity)

↳ Take step in direction of velocity instead of only of loss gradient

- This solves :

+, saddle / minima points since even though gradient = 0, there's still a velocity vector at ~~this locat~~ these points

+, noisy gradients since the noise is "smoothed out" by the ~~a~~ accumulating moving average

+, poor conditioning since ~~a~~ averaging ~~th~~ previous gradients smoothed out the high sensitivity in some directions.

5,

- The advantage of NAG over simple ~~mo~~ momentum is that NAG momentum ~~en~~ has stronger theoretical convergence guarantees and ~~wor~~ consistently works slightly better in ~~practice~~ practice.
- The term "accelerated" in NAG means that for a large enough learning rate, NAG allows larger decay rate α while preventing oscillations.

6,

- First strategy is step decay where we reduce/decay learning rate by some factor (ex: $lr \leftarrow \frac{lr}{2}$) after k iterations. If we plot ~~a graph~~ learning rate vs iterations, the graph is a step function graph ~~this~~, hence the name
- 2nd is exponential decay ~~is~~ where we decay the learning rate ~~by~~ at an exponential rate
  ↳ $Lr \leftarrow \frac{lr}{e^{k/t}}$ where k = decay rate and t = iteration index
- 3rd is fractional decay where we decay by a factor of $1+kt$.
  ↳ $Lr \leftarrow \frac{Lr}{(1+kt)}$, this is smoother than step decay

7,

- Instead of specifying the learning rate like other learning algorithm, newton's method computes the learning rate using hessian matrix as follows:

We want θ such that $\nabla J(\theta) = 0$.

+, Newton's method states that we can start with some initial guess θ_0 ~~and then keep finding an update~~ (which is a vector)

↳ Then we find an update $\Delta\theta$ ~~such that~~ where $\nabla J(\theta_1) = \nabla J(\theta_0 + \Delta\theta)$ brings us closer to 0.

→ keep finding such update $\Delta\theta$ until ~~conve~~ termination ($\nabla J(\theta) = 0$ or close to 0).

+, To find $\Delta\theta$, we can use taylor series expansion: $f(x_0) + \Delta x f'(x_0) = 0$

~~$f(x_0) + \Delta x f$~~

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

$\nabla(\nabla J(\theta_0))$ = Hessian matrix

$$= \begin{bmatrix} \frac{\partial J^2}{\partial\theta_{01}\partial\theta_{01}} & \cdots & \frac{\partial J^2}{\partial\theta_{0n}\partial\theta_{01}} \\ \vdots & & \vdots \\ \frac{\partial J^2}{\partial\theta_{0n}\partial\theta_{01}} & \cdots & \frac{\partial J^2}{\partial\theta_{0n}\partial\theta_{0n}} \end{bmatrix}$$

where $\theta_0 = \cancel{[\theta_1 \ldots \theta_n]}\,[\theta_{01} \ldots \theta_{0n}]$

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

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

↳ Our update is $\theta_1 = \theta_0 + \Delta\theta$

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

↳ This ~~looks like~~ is our update equation for learning using GD.

$\rightarrow$ $\boxed{\text{Learning rate} = H^{-1}}$

- Hessian matrix is a matrix consisting of all second order partial derivatives of a function $f(x, y, \ldots)$ of f. Thus, in the context of newton's method, Hessian matrix is used to determine how big a step ~~we~~ needed to get to point of minima/~~maxima~~ ~~and~~ since the function we're trying to get to the minimum of is already in the first-derivative form. That is, we want θ such that $\nabla J(\theta) = 0$.

  $\rightarrow \nabla(\nabla J(\theta))$ ~~wi~~ $= H$ will give us info on the step we need to take.

8,

- Condition ~~number~~ ~~is used~~.

~~In the context of Newton's method, Condition number ment is used to measure how sensitive the hessian matrix H is to changes in input.~~

- In context of Newton's method, Condition number of Hessian matrix measures sensitivity by the ratio between the largest and smallest singular values $= \frac{SV_1}{SV_m}$ $\rightarrow$ largest / $\rightarrow$ smallest

~~If~~ .

- There's poor conditioning when the conditioning numbers is large and this can lead to 2 problems:

  1, For GD and other first order optimization methods, the loss function will have contours the "stretch" out to very long ellipsoid instead of circular
  → Gradients can point to directions that may not be the best way to descend.
  → Slower

  2, For Newton's method, learning rate = H^{-1}, However, if H is poorly conditioned then H^{-1} can be numerically ~~unstable~~ unstable since the smallest singular values will ~~incr~~ go to large numbers upon inversion (ex: $10^{-10} \to 10^{10}$ after inversion)

9,

Adagrad "approximates" H^{-1} using preconditioner B:

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

↳ $B^{(i)}$ is a scale factor which is the sum of square of previous gradient

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

We keep adding up the past gradient info over iterations

↳ We have an approximation of how fast the loss is changing ⟷ an approximation of H^{-1}

10,

- Problem with Adagrad is that ~~th~~ we normalize by elementwise sum of squares (accumulation of past gradients)

↳ There will cases where learning rate/step size ~~decrease~~ decreases very fast and becomes smaller and smaller.

- RMSprop addresses this by using a decay factor when adding new ~~grad~~ gradients to the gradient sum.

11,

- The update rule for ~~a~~ RMSprop is as follow:

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

$$\theta_j^{(i+1)} \leftarrow \theta_j^{(i)} - \eta \nabla_j L(\theta^{(i)}) \frac{1}{s_j^{(i+1)} + \underbrace{\varepsilon}_{\text{error term}}}$$

⇒ In ~~Adam,~~ Adam, we ~~to~~ incorporate momentum ~~$\nabla L(\theta)$~~ by replacing $\nabla L(\theta)$ by a ~~momentum~~ momentum included term, and make some small changes to the scaling term of RMSprop:

$$m_1^{(i+1)} = \beta_1 \cdot m_1^{(i)} + (1-\beta_1) \nabla L(\theta^{(i)})$$ → ~~ter mome~~ velocity with momentum to replace $\nabla L(\theta)$

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

↳ elementwise step size scale similar to RMSprop.

→ Instead of the RMSprop update, we have a different update equation which includes momentum:

$$\cancel{m_1^{(i+1)}} \cdot \theta^{(i+1)} \leftarrow \theta^{(i)} - \eta\, m_1^{(i+1)} \odot \frac{1}{\sqrt{m_2^{(i+1)}} + \epsilon}$$

- We need a bias ~~correct~~ Corrected term because the moments ~~are~~ m_1, m_2 are initialized to close to 0, so dividing by the 2nd moment m_2 will lead to a large step size.
  ↳ We create a bias corrected term by dividing the moment by a ~~mat~~ number depending on iterations.
  → initial moments ~~are~~ are larger.

12,

- Intuition of GD ~~with Line Search~~ with line search is that we find the "best" step size instead of having a fixed step size everytime. To do this, we basically solve another optimization problem ~~in~~ for the step size.

Given a direction: $u = \nabla f(x)$

↳ We want $\eta^* = \underset{\eta}{\operatorname{argmin}}\, f(x + \eta u)$ = "best" step size

To solve for η^*, we can use ~~line search or~~ bracketing

~~line search: Successive +, Gr~~ General GD with line search:

Start with θ_0 and direction set $\{u^{(i)}\}$

Then:

$$\eta^{(i)} = \underset{\eta}{\operatorname{argmin}}\, f(\theta^{(i)} + \eta\, u^{(i)}) \rightarrow \text{line search step}$$

$$\theta^{(i+1)} \leftarrow \theta^{(i)} + \eta^{(i)} u^{(i)}$$

+, Bracketing algorithm : a strategy ~~to~~ ~~to~~ which determines successively smaller intervals / brackets that contain a root. The algorithm continues until bracket is small enough.

Say we have a bracket $[a, b, c]$ and we have the midpoint between b and c : $x = \frac{b+c}{2}$.

Then we test :

- If $f(x) \leq f(b)$ → we know ~~the~~ we ~~go~~ are in a correct direction and ~~got~~ just got closer to the root
  → we continue search in $[b, x, c]$
- If $f(x) > f(b)$ → we ~~know~~ are ~~in~~ in a wrong direction and just got further from the root
  → we continue search on opposite ~~direct~~ bracket $[a, b, x]$.

↳ Continue until bracket is small enough.

- Alternative to bracketing: explicit computation where we explicitly compute the minima ~~[illegible]~~.

~~bracketing~~ ~~Advantage~~ ~~Disadvantage~~

Bracketing compared to:

~~Bracketing~~: Explicit computation:

+, Advantage: more accurate computation

+, Disadvantage: more costly in terms of computation and cannot perform explicitly computation in some cases (for ex: function is intractable)

13,

- Quasi newton first computes quasi Newton direction which will be the update for the parameter (acts ~~as~~ like the gradient):

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

+, Then we determine the step size η^*, we can either use a fixed step size or an adaptive one (like using ~~line~~ bracketing line search).

+, Then we proceed to update the parameters:

$$\theta^i \leftarrow \theta^{i-1} + \eta^* \Delta\theta$$

+, Then we compute the updated ~~Hessian~~ approximation of Hessian or directly compute updated $(H^{(i)})^{-1}$

To compute the above, we can use BFGS algorithm since it's simpler and less expensive computationally.

+, BFGS. $\Delta\theta = \theta^i - \theta^{i-1}$

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

$$H^i = H^{i-1} + \frac{\Delta J \Delta J^T}{\Delta J^T \Delta\theta} - \frac{H^{(i-1)} \Delta\theta \Delta\theta^T H^{i-1}}{\Delta\theta^T H^{i-1} \Delta\theta}$$

$$\rightarrow (H^i)^{-1} = \left(I - \frac{\Delta\theta \Delta J^T}{\Delta J^T \Delta\theta}\right)(H^{i-1})^{-1}\left(I - \frac{\Delta J \Delta\theta^T}{\Delta J^T \Delta\theta}\right) + \frac{\Delta\theta \Delta\theta^T}{\Delta J^T \Delta\theta}$$

- BFGS advantage over Newton is that it is less computationally expensive because it does not compute the exact ~~Hessia~~ Hessian inverse, but it computes an approximation of Hessian inverse using gradient evaluation.

- Compared with Adam, BFGS:

Advantage: ~~perform~~

+, perform quite well in present of saddle point and flat valley

+, Able to avoid large conditioning number for H (so poor-condition)

Disadvantage:

+, Inaccurate ~~so~~ when using mini batch

→ Require a large amount of / entire dataset.

Regularization:

1,

- We have: $J(\theta) = \frac{1}{m} \sum L_{\theta}(f(x^{(i)}, \theta), y^{(i)}) + \lambda R(\theta)$

Suppose $R(\theta)$ = L1 regularization.

$\rightarrow \frac{\partial J}{\partial \theta} = \frac{\partial}{\partial \theta} \frac{1}{m} \sum L((f(x^{(i)}, \theta)\, y^{(i)})) + \lambda \frac{\partial}{\partial \theta} R(\theta)$

Now $\frac{\partial}{\partial \theta} R(\theta) = \frac{\partial}{\partial \theta} \sum_{i} |\theta_i| = \cancel{\text{sing}(\theta)}\ \text{sign}(\theta)$.

→ For our update, we subtract $\lambda \text{sign}(\theta)$

→ As iterations progress, we essentially driving weights to 0

→ decay to 0 → ~~similar~~ related to weight decay

- Same argument if $R(\theta)$ = L2 regularization.

2,

- Early stopping works by stopping the training process based on the performance on validation set. Train the model as normal, then up to some point after improvement, improving the model comes at the espense of increased generalization error (overfit). Early stopping provides guidance on when this occurs.

- Strategies:

  +, Reuse val data by retraining on all data using the number of iterations determined from validations.

  +, Run early stopping and update weights. Then ~~train the wh~~ while validation loss > Subtraining set loss, train the whole ~~traing~~ training set using those parameters values.

3,

Data augmentation is when we add synthetic data into the dataset so that the model is invariant to wide variety of differences. This is particularly useful when there's not enough data
→ model can easily overfit this small set of data.

~~→ Adding synthetic data to increase model's invariance will help prevent overfitting~~

→ Adding synthetic data helps increase variability in training → better generalization and less overfitting.

4,

- Dropout works by dropping out units in fully connected layers with prob $(1-p)$ at each training phase. The removed nodes are reinstated with original weights in later stage.
- Advantage:
  - +, Reduce node interactions (Co-adaptation)
  - +, reduce overfitting and increase training speed
  - +, Reduce dependency on a single node, distribute features across multiple nodes.
- Disadvantage:
  - +, Longer training as not all units are available at each step.

6,

- We take as input a batch of data. This input will be normalized by subtracting mean of batch and divide by variance of batch. → $\mu_B = \frac{1}{m}\sum_{1}^{m} x_i$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

This gives equal importance to features with different scales

Then, we use the normalized input to calculate the outputs ~~by applying a shift and scale factor of the batch~~ by applying a shift and scale factor.

This output is then fed into activation for normal training.

↳ output of batchnorm layer $= \gamma \hat{x}_i + \beta$. where γ and β ~~is~~ are parameters to learn

During testing, we use the avg normalization values computed during training (avg from all training batches).

- BN introduces randomness to training because ~~it~~ the input batches are randomly selected

7,

- We use the scale and shift params because we don't ~~desire~~ want to always have mean 0 and variance 1 for all output, else the value will be too close to 0.
- When scale ~~factor~~ parameters = variance of data and shift ~~factor~~ params = mean of data → normalization will be cancelled

- ~~Initially~~, $\gamma = 1$, $\beta = 0$ are good initial values since this resembles the normalized output.
- ~~Th~~ γ and β can be trained along with our other model parameters. We take the gradient of loss

w.r.t γ and β and update just like other weights.

8,

- Ensemble classifiers use majority votes or average among multiple ~~clas~~ models and thus in effect kind of smooths out the outlier ~~clas~~ overfitted model.

  ↳ ~~Help with~~ Assist with preventing overfitting by averaging the overfitted model/models out with all other models.

- Strategy:

  +, Change input data.

  +, Change parameters of model

  +, For each training phase, record the snapshot of the model.

5,

We have a total of 2^n dropped out networks. ~~Multiplying~~ ~~the out~~ ~~Multiplyin~~. Expected value of all ~~on~~ combinations of dropped out network (2^n) can be approximated efficiently by multiplying the output of each node by p

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

~~Take~~

expectation over <u>all</u> combinations of n nodes of the predictions made by these ~~not~~ combinations