Assignment 3 Solution

Loss:

1. L1: $L_i(\theta) = \sum_{j=1}^{k} |\hat{y}_j^{(i)} - y_j^{(i)}|$
   Advantage: robust since it's not sensitive to outliers

   L2: $L_i(\theta) = \sum_{j=1}^{k} (\hat{y}_j^{(i)} - y_j^{(i)})^2$
   Advantage: stable since the regression parameters are continuous functions of the data.

   Huber:

$$L_\delta(y, f(x)) = \begin{cases} \frac{1}{2}(y - f(x))^2 & \text{for} |y - f(x)| \le \delta, \\ \delta |y - f(x)| - \frac{1}{2}\delta^2 & \text{otherwise.} \end{cases}$$

   Advantage: It curves around the minima which decreases the gradient. And it's more robust to outliers than L2 loss.

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

   Where log(cosh(d)) = $\begin{cases} \frac{1}{2}d^2 & if\ d\ is\ small \\ |d| - log2 & otherwise \end{cases}$

   Advantage: it will not be so strongly affected by the occasional wildly incorrect prediction.

2. Equation: $L_i(\theta) = -\sum_{j=1}^{k} y_j^{(i)} \log(\hat{y}_j^{(i)})$, where $\hat{y}_j^{(i)} = P(y = j \mid x^{(i)})$ j ∈ [1, k]
   Likelihood: $L(\theta) = \prod_{i=1}^{m} \prod_{j=1}^{k} (P(y = j \mid x^{(i)}))^{y_j^{(i)}}$
   We need to maximum log-likelihood which is equal to minimize negative log-likelihood:

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

   The worst cross-entropy loss value for random assignment:
   $P(y = j \mid x^{(i)}) = 1/k \Rightarrow -\log(P(y = j \mid x^{(i)})) = \log k$

3. $L_i(\theta) = -\sum_{j=1}^{k} y_j^{(i)} \log(\hat{y}_j^{(i)})$, where $\hat{y}_j^{(i)} = P(y = j \mid x^{(i)})$ j ∈ [1, k]
   Softmax loss is used for multi-class classification.

4. Equation: $L(\theta) = -\sum_{i=1}^{m} \sum_{j=1}^{k} y_j^{(i)} \log(\frac{y_j^{(i)}}{\hat{y}_j^{(i)}})$

   Kullback-Liebler loss a measure of how one probability distribution is different from a second, reference probability distribution.
   When the class label doesn't change, the Kullback-leibler is the same as cross-entropy.

5. Hinge loss:

   2-class: $L_i(\theta) = \max\left(0, 1 - y^{(i)}\hat{y}^{(i)}\right)$

   Multi-class: $L_i(\theta) = \sum_{j \neq t} \max(0,\ \hat{y}_j^{(i)} - y_t^{(i)} + 1)$

   Squared hinge loss:

   2-class: $L_i(\theta) = \max(0,\ 1 - y^{(i)}\hat{y}^{(i)})^2$

   Multi-class: $L_i(\theta) = \frac{1}{2}\sum_{i=1}^{k} \max(0,\ \hat{y}_j^{(i)} - y_t^{(i)} + 1)^2$

   The fundamental idea behind is that we do not only make correct classification but also introduce a margin to make sure separate the classes as far as possible.

   The worst expected value before learning is k-1 where k is the number of class labels.
   $L_i(\theta)$ = (k-1) * max(0, 0 – 0 + 1) = k -1

6. The prediction table:

	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$
y=1	0.5	1.3	1.4
y=2	0.4	0.8	-0.4
y=3	0.3	-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. The purpose is to get a simpler solution with lower θ which leads to more stable and generalize better.
   L1: R(θ) = $\sum_{i,j} |\theta_{i,j}|$
   L2: R(θ) = $\sum_{i,j} (\theta_{i,j})^2$
   L1 makes weights sparse, L2 makes weights smaller while spreading them.
   Regularization term coefficient is a hyper-parameter. We can start with no regularization.

8. Gradient with L1 loss:
   $$\frac{\partial L}{\partial \theta} = \frac{\partial}{\partial \theta}\frac{1}{m}\sum_{i=1}^{m} L_i(y(x^{(i)},\ \theta),\ y^{(i)}) + \lambda \frac{\partial}{\partial \theta} R(\theta)$$
   L1 adds $\lambda sign(\theta)$ to the gradient.

   Gradient with L2 loss:
   $$\frac{\partial L}{\partial \theta} = \frac{\partial}{\partial \theta}\frac{1}{m}\sum_{i=1}^{m} L_i(y(x^{(i)},\ \theta),\ y^{(i)}) + 2\lambda\theta$$
   L2 adds $2\lambda\theta$ to the gradient.

9. Kernel regularization introduces a weight decay on weight w.
   Bias regularization is to reduce bias b and expects to get smaller output values.
   Activity regularization introduces a decay on output $\hat{y}$ and expects the output value close to zero.

Optimization:

1. Back propagation is easy to compute and uses symbolic derivatives.
   Direct numerical computation of gradients can be used for verification.

2. SGD: update gradient with batch of training dataset.
   GD: update gradient with entire training dataset.
   SGD leads to converge faster because it updates the weights more frequently.

3. Large batch size reduces the variance of the stochastic gradient updates however degrade the quality of model due to it tends to converge to sharp minimizers of the training function.
   Small batch size updates the weights more frequently however may stuck in a locally optimal solution.

   4 main problems with SGD:
   1) How to determine the learning rate?
   2) What if some parameters are more sensitive than the others?
   3) How to avoid getting stack to the local minimal?
   4) Minibatch gradient estimate are noisy

4. Poor conditioning: Smooth out by averaging with previous gradient
   Minimum/saddle points: There is a velocity vector even at a local minimum. It will have a better chance to get out of the saddle points.
   Noisy gradients: Smooth out by moving averagely.

5. NAG momentum is more accurate than simple momentum because it corrects the location for gradient at future step.
   Accelerated: g($v^{(i+1)} - v^{(i)}$)

6. 1) $\eta^{(i+1)} = \frac{\eta^{(i)}}{2}$
   2) $\eta^{(i)} = \eta^{(0)} e^{-k/i}$
   3) $\eta^{(i)} = \eta^{(0)}/(1 + ki)$

7. $\theta_{(i+1)} \leftarrow \theta_{(i)} - H^{-1}\nabla\tau(\theta)$
   Where $H = \nabla(\nabla\tau(\theta))$ and we consider H^{-1} as the learning rate.

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

8. Condition number defines sensitivity using the singular value of the Hessian matrix:
   Condition number = $\frac{largest\ SV}{smallest\ SV}$ (SV: singular value)
   When the condition number is high, matrix is particularly sensitive to error in input which leads to harder to optimize.

9. AdaGrad algorithm approximates the invers of Hessian by computing the matrix
   $B^{(i)} = diag(\sum_{j=1}^{i} \nabla\tau(\theta^{(j)}) \nabla\tau\,(\theta^{(j)})^T)^{1/2}$
   Parameter update: $\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta B^{-1} \nabla\tau(\theta)$

10. In AdaGrad, we normalize by elementwise sum of square gradient, so the step size will be smaller as iteration progress.
    RMSProp uses a decay factor when adding new gradients to the gradient sum.

11. 1) adding velocity with momentum:
    $m_1^{(i+1)} = \beta_1 m_1^{(i)} + (1 - \beta_1) \nabla L(\theta^{(i)})$
    2) adding elementwise step size scale:
    $m_2^{(i+1)} = \beta_2 m_2^{(i)} + (1 - \beta_2)(\nabla L(\theta^{(i)}) \odot \nabla L(\theta^{(i)}))$
    3) final update equation:
    $\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta m_1^{(i+1)} \odot \frac{1}{\sqrt{m_2^{(i+1)} + \epsilon}}$
    The moment is initialized to zero, when dividing by the second moment we will add a large step. Thus, we need to add a bias correction term dividing the momentum by a number depending on iteration number.

12. Line search: Instead of a fixed step size, find the optimal step size given a direction.
    Given direction: $\mu = \nabla f(x)$
    Best step size: $\eta^* = argmin_\eta f(x + \eta\mu)$
    Gradient descent: $\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta^* \nabla f(\theta^{(i)})$

    Bracketing: search for minimum point.
    Give bracket [a, b, c]: x = (b + c)/2
    If f(x) ≤ f(b) ⇒ [b, x, c]
    If f(x) > f(b) ⇒ [a, b, x]
    We repeat this procedure until the bracket is small enough.

    Alternative: coordinate decent.
    Advantage of bracket: more accurate
    Disadvantage of bracket: more computation

13. Quasi-Newton approximates the Hessian inverse using gradient evaluation.
    1) Compute quasi-newton direction: $\nabla\theta = -\left(H^{(i-1)}\right)^{-1}\nabla J(\theta^{(i-1)})$
    2) Determine step size η^*
    3) Compute parameter update: $\theta^{(i)} \leftarrow \theta^{(i-1)} + \eta^*\nabla\theta$
    4) Compute updated Hessian approximate $H^{(k)}$

    Advantage of the BFGS algorithm over Newton: the cost is O(n^2) which is more efficient than the Newton cost O(n^3).

    Advantages of BFGS compared with Adam: more accurate

    Disadvantages of BFGS compared with Adam: compute with entire dataset (computation consuming)

Regularization:

1. Multiply each coefficient by p $\in$ [0, 1]. As iterations progress, weights that are not reinforced decay to zero. This is equivalent to add regularization term to loss function.

2. Early stop: Stop when validation error increase instead of when training error stop decrease. So it can prevent the network overfit with training data.
   Strategy 1: Retrain on all data using the number of iterations determined from validation.
   Strategy 2: Continue training from previous weight with entire data when validation loss is bigger than training loss.

3. 1) Augment in feature or data domain
   2) Augment by interpolating between examples or by adding noise
   3) Augment by transforming data

   Data augmentation can increase variability in the training data which results to better generalization and prevent overfitting.

4. At each training stage dropout units in fully connected layers with probability (1-p), where p is hyper-parameter. Removed nodes are reinstated with original weights in the subsequent stage.

   Advantage:

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

Disadvantage:

Longer training due to dropout. Not all units are available at each stage.

5. We can multiply the output of each node by P is equivalent to computing expected value for 2^n dropped out networks.

$$\hat{y} = E_D(\mathrm{f(x, D)}) = \int \mathrm{P(D)F(x, D)}\ \mathrm{dD}$$

6. Training:

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Algorithm 1: Batch Normalizing Transform, applied to activation x over a mini-batch.

Testing: use the average normalization values computed during training.

The randomness is introduced by the input batch.

7. Some saturation is needing to terminate learning. So, we may need to scale a shift after normalization. We can control how much normalization will do.

$\gamma_j = \sigma_j$ and $\beta_j = \mu_j$ causes the normalization to be canceled.

Calculate the gradient with respect to γ and β to learn.

Good initial value: $\gamma_j = 1, \beta_j = 0$.

8. With training multiple independent models and using majority vote or average during testing, we can reduce overfitting because our prediction is an average which is less sensitive to bias of the one single model.

Strategies: Change data; Change parameters; Record multiple snapshots of the model during training; By random drop out