CS577: Assignment 3

Jane Downer
Department of Computer Science
Illinois Institute of Technology

October 25, 2022
(used 2 late days)

Loss

1. L1 L2, Huber loss, and Log-cosh are all loss functions for regression.

L_1:	$L_i(\theta) = \sum_{j=1}^{k} \lvert \hat{y}_j^{(i)} - y_j^{(i)} \rvert$	Small penalty on loss
L_2:	$L_i(\theta) = \sum_{j=1}^{k} \left(\hat{y}_j^{(i)} - y_j^{(i)}\right)^2$	Larger penalty on loss – more sensitive to outliers
$Huber$:	$L_i(\theta) = \sum_{j=1}^{k} \rho_\delta\left(\hat{y}_j^{(i)} - y_j^{(i)}\right)$ where $\rho_\delta(d) = \begin{cases} \frac{1}{2}d^2 & if\ \lvert d \rvert \le \delta \\ \delta\left(d - \frac{1}{2}\delta\right) & otherwise \end{cases}$	In between L1 and L2 – quadratic for small d and linear for large d. Caps the loss of outliers.
$Log - cosh$:	$L_i(\theta) = \sum_{j=1}^{k} \log\left(\cosh\left(\hat{y}_j^{(i)} - y_j^{(i)}\right)\right)$ where $\log(\cosh(d)) = \begin{cases} \frac{1}{2}d^2 & if\ d\ is\ small \\ \lvert d \rvert - \log(2) & otherwise \end{cases}$	Has an effect similar to Huber loss: reduces sensitivity to outliers.

2. Cross-entropy loss is used for classification. With classification we want to maximize likelihood, given as:

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

This is equivalent to minimizing negative log likelihood, given as:

$$\begin{aligned} l(\theta) \;&= -\sum_{i=1}^{m} \sum_{j=1}^{k} y_j^{(i)} \cdot \log\left(P\left(y = j \mid x^{(i)}\right)\right) \\ &= -\sum_{i=1}^{m} \sum_{j=1}^{k} y_j^{(i)} \cdot \log\left(\hat{y}_j^{(i)}\right) \end{aligned}$$

Cross-entropy loss for the i^{th} example:

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

In the worst case, the ground truth is 1 and the predicted output is as close to 0 as possible (without actually being 0), in which the loss value will be infinity.

3. Softmax loss is softmax activation in the output layer plus cross-entropy loss. Softmax loss for i^{th} example can be represented as:

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

Where f is the softmax function and $\hat{y}$ is the output of the final layer.

4. Kullback-Liebler loss is a measurement of the distance between distributions. This is represented as:

$$L(\theta) \;=\; \sum_{i=1}^{m} \sum_{j=1}^{k} y_j^{(i)} \cdot \log\left(\frac{y_j^{(i)}}{\hat{y}_j^{(i)}}\right)$$

Where the loss for the i^{th} example is:

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

This is equivalent to:

$$\begin{aligned} L_i(\theta) &= \sum_{j=1}^{k} y_j^{(i)} \cdot \log\left(y_j^{(i)}\right) - \sum_{j=1}^{k} y_j^{(i)} \cdot \log\left(\hat{y}_j^{(i)}\right) \\ &= -entropy + cross\text{-}entropy \end{aligned}$$

If $y^{(i)}$ does not change / entropy is 0, KL divergence is equivalent to cross-entropy.

5. In a **binary** classification problem, Hinge Loss corresponds to a 2-class SVM. The two classes are represented as either 1 or -1, and the loss is determined by (a) whether the ground truth and predicted class have the same sign, and (b) whether the predicted value falls within the SVM margin. Assuming a margin of 1:

$$Hinge\ Loss\ (binary): \qquad L_i(\theta) \;= \max\left(0, 1 - y^{(i)} \cdot \hat{y}^{(i)}\right)$$

In a **k-class** classification problem, Hinge Loss corresponds to an extension of the binary SVM to a k-class problem. In this case, loss is zero only if the distance from the ground truth boundary is greater than the distance from the boundaries of the incorrect classes plus a margin. Assuming a margin of 1:

$$Hinge\ Loss\ (k-class): \qquad L_i(\theta) \;= \sum_{j \neq t} \max\left(0, \hat{y}_j^{(i)} - y_t^{(i)} + 1\right)$$

In practice, for k-class classification, we sum over all values from 1 to k (not just the incorrect classes), meaning we use the definition below. This means all values of loss are increased by 1, which doesn't matter, since we are minimizing it, which is unaffected by this added constant.

$$Hinge\ Loss\ (k-class): \qquad L_i(\theta) \;= \sum_{j=1}^{k} \max\left(0, \hat{y}_j^{(i)} - y_t^{(i)} + 1\right)$$
$$(redefined)$$

Squared hinge is the above function, squared and divided by 2. The addition of the exponent means this will be more sensitive to outliers and loss values will be higher.

$$Squared\ Hinge\ Loss\ (k-class): \qquad L_i(\theta) \;= \frac{1}{2}\sum_{j=1}^{k} \max\left(0, \hat{y}_j^{(i)} - y_t^{(i)} + 1\right)^2$$

6.

$$\begin{aligned} y^{(1)} = 1 \qquad & \hat{y}^{(1)} = (0.5, 0.4, 0.3) \qquad & L^{(1)} &= \max(0,\ 0.4 - 0.5 + 1) + \max(0,\ 0.3 - 0.5 + 1) \\ & & &= 0.9 + 0.8 \\ & & &= 1.7 \\ y^{(2)} = 2 \qquad & \hat{y}^{(2)} = (1.3, 0.8, -0.6) \qquad & L^{(2)} &= \max(0,\ 1.3 - 0.8 + 1) + \max(0, -0.6 - 0.8 + 1) \\ & & &= 1.5 + 0 \\ & & &= 1.5 \\ y^{(3)} = 3 \qquad & \hat{y}^{(3)} = (1.4, -0.4, 2.7) \qquad & L^{(3)} &= \max(0,\ 1.4 - 2.7 + 1) + \max(0, -0.4 - 2.7 + 1) \\ & & &= 0 + 0 \\ & & &= 0 \end{aligned}$$

7. By adding regularization, we modify the loss term to include a preference for smaller models – i.e., models with smaller weights – because these models generalize better and are less likely to overfit. The regularization term

measures how large the weights are, and by adding it to the loss term, we have the objective of both minimizing the loss and minimizing the weights.

$$L_1\ regularization: \qquad R(\theta) = \sum_{i,j} \left|\theta_{i,j}\right|$$
$$L_2\ regularization: \qquad R(\theta) = \sum_{i,j} \theta_{i,j}^2$$

L_2 regularization's contribution to the loss function is the sum of squared weights, which is smallest when the individual weights are more evenly distributed at values between 0 and 1. L_1 regularization's contribution to the loss function is simply the summed absolute values of the weights, making it more likely to push some weights to 0, resulting in a sparse weight matrix.

The regularization term λ is a hyperparameter that reflects how much we care about regularization. A larger λ means the effect of regularization will be stronger.

8. Adding L_1 and L_2 to the loss function results in weight decay equivalent to $\lambda \cdot \frac{\partial R(\theta)}{\partial \theta}$, which is subtracted from θ during gradient descent.

$$\begin{aligned} for\ L_1\ regularization: \qquad \frac{\partial R(\theta)}{\partial \theta} &= \frac{\partial}{\partial \theta}\left(\sum_i |\theta_i|\right) \\ &= sign(\theta) \\ &\boldsymbol{weight\ decay = \lambda \cdot sign(\theta)} \end{aligned}$$

$$\begin{aligned} for\ L_1\ regularization: \qquad \frac{\partial R(\theta)}{\partial \theta} &= \frac{\partial}{\partial \theta}\left(\sum_i (\theta_i)^2\right) \\ &= \frac{\partial}{\partial \theta}\left(\theta^T \theta\right) \\ &\boldsymbol{weight\ decay = \lambda \cdot \theta} \end{aligned}$$

9. Kernel regularization, bias regularization, and activity regularization are regularization options in Keras. Kernel regularization is the regularization/reduction of W, bias regularization is the regularization/reduction of b, and activity regularization is the regularization/reduction of $\hat{y}$ (and, consequently, W and b).

   We normally just use kernel regularization. If the function is expected to output small values, we may use bias regularization. And if the function is expected to output values close to zero, we may use activity regularization.

Optimization

1. Numerically computing the gradient involves using the definition of a partial derivative:

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

   This is easy to do, but slow and computationally expensive. Backpropagation is also easy, and makes computation easier by allowing us to use symbolic derivatives.

2. With gradient descent, we perform an update after viewing all the examples. With stochastic gradient descent, the update occurs after processing one example or a subset (minibatch) of examples, which is faster to converge because there are more frequent updates.

3. The tradeoff is between speed and accuracy. A smaller batch size means more frequent updates and faster convergence, however making updates based on a smaller batch means the updates are less accurate. A larger batch size means less frequent updates and slower convergence, but the updates are more accurate because they are based on the entire dataset.

   Stochastic gradient raises four problems:

   - What should the learning rate be?
   - What if loss is more sensitive to some parameters than others? If all the parameters don't behave in the same way, they shouldn't have the same learning rates – but it is often impossible to find individual learning rates, for example, in the case that there are millions of parameters.

- How do we avoid getting stuck and local minima or saddle points, where the gradient is zero but loss is not minimized?
- Minibatch gradient estimates are noisy, since the updates are less stable than if we used the entire dataset.

4. How SGD with momentum updates parameters in the following way:

ρ: scales the velocity. If $\rho = 0$, this is the same as ordinary SGD.

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

Or, rewritten:

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

Which is equal to:

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

Where $\tilde{\rho} = -\eta \cdot \rho$.

- Poor conditioning: if loss is more sensitive to some parameters than others, SGD with momentum helps by averaging the next gradient with previous ones.
- Local minima/saddle points: even when the gradient is zero, there is still a velocity, which allows updates to continue.
- Noisy gradients: even if two consecutive gradients are in completely different directions, SGD with momentum makes updates that are an average between them, which smooths them out.

5. Nesterov Accelerated Gradient (NAG) uses the following update rules:

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

If $\boldsymbol{\tilde{\theta}^{(i)}} \equiv \theta^{(i)} + \rho \cdot v^{(i)}$, this is equivalent to:

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

Since $\theta^{(i)} = \tilde{\theta}^{(i)} - \rho \cdot v^{(i)}$, then we can rewrite $\theta^{(i+1)} \leftarrow \theta^{(i)} + v^{(i+1)}$ as:

$$\left(\tilde{\theta}^{(i+1)} - \rho \cdot v^{(i+1)}\right) \leftarrow \left(\tilde{\theta}^{(i)} - \rho \cdot v^{(i)}\right) + v^{(i+1)}$$

Which simplifies to:

$$\tilde{\theta}^{(i+1)} \leftarrow \tilde{\theta}^{(i)} + v^{(i+1)} + \rho \cdot \left(v^{(i+1)} - v^{(i)}\right)$$

In we let $\theta^{(i)} = \tilde{\theta}^{(i)}$, then we arrive at the following formulas:

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

The advantage of NAG over SGD with momentum is that we operate according to the future time step rather than the current one, which allows us to make updates based on our best guess at what the future will look like. We say that NAG is "accelerated" because it incorporates the change in velocity (scaled by ρ).

6. Strategies for learning rate decay, where k is the 'decay rate' and t is the iteration index:
   - Step decay: $\eta \leftarrow \frac{\eta}{2}$
     - Produces abrupt change in learning rates

- Exponential decay: $\eta \leftarrow \eta_0 \cdot e^{-k/t}$
  - Produces gradual change in learning rates
- Fractional decay: $\eta \leftarrow \eta_0/(1+kt)$
  - Another way to produce gradual change in learning rates

7. Newton's method:
   1. Find $\boldsymbol{x}$ such that $\boldsymbol{f(x) = 0}$
   2. Start with guess $\boldsymbol{x_0}$
   3. Find update $\boldsymbol{\Delta x}$ such hat $\boldsymbol{f(x_0 + \Delta x) = 0}$

   We can find Δx using a Taylor series expansion. If the function f takes a scalar as input and produces a scalar as output, we can define $f(x_0 + \Delta x)$ as:

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

   In vector form, this can be defined as:

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

   We want $f(x_0) + \Delta x^T \Delta f(x_o)$ to equal 0. Replacing f with ∇J and x with θ, our objective is:

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

   $\nabla(\nabla J(\theta_0))$ is equivalent to J's Hessian matrix, H:

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

   Using H in place of $\nabla(\nabla J(\theta_0))$, we can rewrite our objective:

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

   To satisfy that condition, $\Delta\theta$ must equal $H^{-1} \cdot \nabla J(\theta_0)$. With this value, we can iteratively update θ until our objective is met:

$$\theta^{(i+1)} \;\leftarrow\; \theta^{(i)} - H^{-1} \cdot \nabla J(\theta_0)$$

   This looks like gradient descent, expect now a specified learning rate is replaced with H^{-1}, which we can compute directly.

8. The condition number is the ratio of the Hessian matrix's largest singular value to its smallest singular value. Singular values are used to measure the sensitivity of the loss with respect to the parameters. A large condition number means loss is much more sensitive to some parameters than others, and that different features need different learning rates. This means the magnitude of the condition number can indicate how much we need the Hessian matrix and Newton's Method.

9. Adagrad approximates the Hessian with the matrix B.
Simplify notation by letting $J_{\theta j} = J(\theta^{(j)})$. Then B is given by:

$$\begin{aligned} B^{(i)} &= diag\left(\textstyle\sum_{j=1}^{i} \nabla J_{\theta j} \nabla J_{\theta j}{}^T\right)^{1/2} \\ B^{(i+1)} &= diag\left(B^{(i)} + \nabla J_{\theta(i+1)} \nabla J_{\theta(i+1)}{}^T\right)^{1/2} \end{aligned}$$

The parameter updates will be scaled by a factor $s_j^{(i+1)}$ – the squared sum of past gradients. Its update rule is:

$$s_j^{(i+1)} \quad \leftarrow \; s_j^{(i)} + \left\|\nabla_j L\left(\theta^{(i)}\right)\right\|^2 \qquad \text{where } \nabla_j L\left(\theta^{(i)}\right) = \frac{\partial J}{\partial \theta_j}\left(\theta^{(i)}\right)$$

The parameter update itself is:

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

When the sum of squared past gradients is large in a particular direction, then the update in that direction will be smaller – in this sense, it approximates the Hessian

10. The sum of squared past grad gradients increases with each iteration, and as a result, the updates will get smaller and smaller. RMSProp addresses this problem by scaling the summed previous gradients with weight ρ and scaling the new gradient with weight $(1-\rho)$ when summing the two terms.

    Again, simplify notation by letting $J_{\theta j} = J\left(\theta^{(j)}\right)$. $\tilde{B}$ is given by:

$$\tilde{B}^{(i+1)} \quad = \quad diag\left(\rho \cdot \sum_{j=1}^{i} \nabla J_{\theta j} \nabla J_{\theta j}{}^{T} + (1-\rho) \cdot \nabla J_{\theta(i+1)} \nabla J_{\theta(i+1)}{}^{T}\right)^{1/2}$$

    The update formulas are:

$$\begin{cases} s_j^{(i+1)} \leftarrow \gamma \cdot s_j^{(i)} + (1-\gamma) \cdot \left\|\nabla_j L\left(\theta^{(i)}\right)\right\|^2 \\ \theta_j^{(i+1)} \leftarrow \theta_j^{(i)} - \eta \cdot \nabla_j L\left(\theta^{(i)}\right) \cdot \frac{1}{\sqrt{s_j^{(i+1)} + \epsilon}} \end{cases}$$

11. Consider SGD with momentum and RMSProp:

SGD with momentum: $\begin{cases} v^{(i+1)} \leftarrow \rho \cdot v^{(i)} + \nabla L\left(\theta^{(i)}\right) \\ \theta^{(i+1)} \leftarrow \theta^{(i)} - \eta \cdot v^{(i+1)} \end{cases}$ (take step in direction of velocity, which is the old velocity averaged with past gradients)

RMSProp: $\begin{cases} s_j^{(i+1)} \leftarrow \gamma \cdot s_j^{(i)} + (1-\gamma) \cdot \left\|\nabla_j L\left(\theta^{(i)}\right)\right\|^2 \\ \theta_j^{(i+1)} \leftarrow \theta_j^{(i)} - \eta \cdot \nabla_j L\left(\theta^{(i)}\right) \cdot \frac{1}{\sqrt{s_j^{(i+1)} + \epsilon}} \end{cases}$ (scale step size by a function of the squared sum of gradients)

With Adam, the step size is a function of the first and second moment – the first moment incorporates the idea of momentum, and the second moment incorporates the idea of scaling the step size (as in RMSProp).

$\odot$: element-wise matrix multiplication

$$\begin{cases} m_1^{(i+1)} = \beta_1 \cdot m_1^{(i)} + (1-\beta_1) \cdot \nabla L\left(\theta^{(i)}\right) \\ m_2^{(i+1)} = \beta_2 \cdot m_2^{(i)} + (1-\beta_2) \cdot \left(\nabla L\left(\theta^{(i)}\right) \odot \nabla L\left(\theta^{(i)}\right)\right) \\ \theta^{(i+1)} \leftarrow \theta^{(i)} - \eta \cdot m_1^{(i+1)} \cdot \frac{1}{\sqrt{m_2^{(i+1)} + \epsilon}} \end{cases}$$

The moments m_1 and m_2 are initialized to 0, so when we estimate $\theta^{(i+1)}$, we divide by a small number $\left(\sqrt{m_2^{(i+1)}} + \epsilon\right)$ which results in a large step size. We use a bias correction term to fix this. We divide the moments m_1 and m_2 by a value that depends on the iteration number i – in this way we can ensure that initial moments sized in a way that the step size is reasonable.

$$\tilde{m}_1^{(i+1)} = \frac{m_1^{(i)}}{1 - \beta_1^i}$$
$$\tilde{m}_2^{(i+1)} = \frac{m_2^{(i)}}{1 - \beta_2^i}$$
$$\theta^{(i+1)} \leftarrow \theta^{(i)} - \eta \cdot \tilde{m}_1^{(i+1)} \cdot \frac{1}{\sqrt{\tilde{m}_2^{(i+1)}} + \epsilon}$$

12. For gradient descent with line search, we are given a direction/gradient ($U = \nabla f(x)$). We look for the learning rate η that minimizes the resulting loss ($f(x + \eta \cdot U)$) along the line defined by this direction.

    Best step size: $\eta^* = \underset{\eta}{\operatorname{argmin}}(f(x + \eta \cdot U))$

    Bracketing:

    Given bracket $[a, b, c]$:

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

    if $f(x) \leq f(b) \rightarrow$ redefine bracket by $[b, x, c]$
    if $f(x) > f(b) \rightarrow$ redefine bracket by $[a, b, x]$

    Repeat this process until the bracket is small enough.

    An alternative is to gradient descent with line search / bracketing is to use coordinate search, which defines our direction set $\{U^{(i)}\}$ as a series of horizontal and vertical steps along the coordinates. Bracketing produces optimal results, but coordinate descent is easier because there's no optimization problem.

13. Quasi-Newton methods try to approximate the inverse of the Hessian matrix. One such method is BFGS, which iteratively updates $\left(H^{(i)}\right)^{-1}$.

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

    The update rule for $\left(H^{(i)}\right)^{-1}$ is:
    $$\left(H^{(i)}\right)^{-1} \leftarrow \left(I - \frac{\Delta\theta \Delta J^T}{\Delta J^T \Delta\theta}\right)\left(H^{(i-1)}\right)^{-1}\left(I - \frac{\Delta J \Delta\theta^T}{\Delta J^T \Delta\theta}\right) + \frac{\Delta\theta\Delta\theta^T}{\Delta J^T \Delta\theta}$$

    Advantages of BFGS over Newton: lower complexity because we don't need to invert the Hessian – $O(n^2)$ rather than $O(n^3)$. BFGS is more computationally expensive than Adam, so it is less practical to use, but will eventually achieve lower loss and higher accuracy.

Regularization

1. Weight decay involves multiplying each coefficient by a decay rate $\rho \in [0,1]$. This has the effect of reducing weights that are not reinforced to 0. We saw earlier that we can reduce some weights to zero (or close to zero) by adding a regularization term to the loss function – in this sense, the two methods have the same effect.

2. We can tell that overfitting is happening when the validation loss begins to increase during training. With early stopping, we split the training set into a training subset and a validation set. We train on the subset and stop when validation error starts to increase. There are two strategies to take from this point:

1. Take note of the number of iterations that elapse before validation error increased. Train the entire training set (rather than just the subset) for this number of iterations.
2. Take note of the loss on the training subset after early stopping (call this value ϵ). Using the updated weights that we obtain at the time of early stopping, we continue training, but adding data, and using the entire training set rather than just the training subset. We continue training until the validation loss is no longer greater than ϵ.

3. Data augmentation prevents overfitting by making the training set larger and more varied, which improves its ability to generalize and makes it more robust to changes in the inputs. We can augment the data by interpolating between examples and adding noise. We can augment image data by transforming the data (crop, rotate, scale, intensity, etc.), which helps introduce scale, illumination, and rotation invariance.

4. Dropout is used during training. It involves applying 0 weight to individual units in a fully-connected layer with probability $(1 - p)$, where p is a hyperparameter. In the next stage, we reinstate the original weights. An advantage of dropout is that it decreases interaction effects between nodes (co-adaptation), and reduces overfitting by making the output less dependent on individual nodes. It also decreases the number of nodes we need to update during backpropagation. The disadvantage is that we have less information so it takes longer to converge.

5. Dropout is not used during testing. As a result, we expect output from all units, since none are dropped, meaning the summed values of the output will be larger. If we multiply the output of each node by p (the probability that a node would be dropped during training), we effectively compute the expected value for the 2^n possible dropped-out networks (2^n because there are n nodes, each of which has two options – drop or do not drop). This is the value we expect from an ensemble of the 2^n models, all of which share the same parameters.

$$\hat{y} = E_0[f(x, D)] = \int p(D) \cdot f(x, D)\ dD$$

Alternatively, rather than multiplying outputs by p during prediction, we can multiply outputs by $1/p$ during training – this reduces the number of total operations.

6. For each example i in a batch, a given layer will produce one output $z_j^{(i)}$ for every unit j. In batch normalization, we normalize the outputs using separate methods for training and inference. During training:

$$\hat{z}_j^{(i)} = \frac{z_j^{(i)} - \mu_j}{\sigma_j}$$ ← normalized value for the j^{th} output of example i in a batch

Where:

$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} z_j^{(i)}$$ ← average output of j^{th} unit across all m examples in the batch

$$\sigma_j = \sqrt{\left(\frac{1}{m}\sum_{i=1}^{m}\left(z_j^{(i)} - \mu_j\right)^2\right)}$$ ← standard deviation of unit j output across all m examples in the batch

During inference, we normalize using the average the normalization values from all training batches.

Because batches are random, batch normalization incorporates randomness into the training process.

7. Scale and shift parameters allow us to obtain some saturation if necessary (e.g., the saturation needed to for the network to converge). If batch normalization is not necessary, the network will learn to set the scale factor (γ_j) equal to standard deviation (σ_j) and the shift factor (β_j) equal to the mean (μ_j), which will effectively cancel the effect of batch normalization.

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

If $\gamma_j = \sigma_j$ and $\beta_j = \mu_j$:

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

We can initialize the scale parameter to 1 (or ones) and the shift parameter to 0 (or zeros) – this is the same as having pure batch normalization. If we start with these values, the network can adjust them if learns it needs to.

8. An ensemble classifier involves training multiple independent models, and the output is either the majority vote (if classification) or average (if regression or prediction) of every models' outputs. By taking the average or the majority vote, we can get rid of (or reduce) the effects of any individual model's overfitting.