Regularization and Optimization of Backpropagation

Keith L. Downing

The Norwegian University of Science and Technology (NTNU)
Trondheim, Norway
keithd@idi.ntnu.no

February 25, 2020

Outline

- Regularization
- Optimization

Regularization

Definition of Regularization

Reduction of testing error while maintaining a low training error.

- Excessive training does reduce training error, but often at the expense of higher testing error.
- The network essentially **memorizes** the training cases, which hinders its ability to **generalize** and handle new cases (e.g., the test set).
- Exemplifies tradeoff between bias and variance: an NN with low bias (i.e. training error) has trouble reducing variance (i.e. test error).
- Bias$(\tilde{\theta}_m) = E(\tilde{\theta}_m) - \theta$
- Where θ is the parameterization of the true generating function, and $\tilde{\theta}_m$ is the parameterization of an estimator based on the sample, m. E.g. $\tilde{\theta}_m$ are weights of an NN after training on sample set m.
- Var$(\tilde{\theta}_m)$ = degree to which the estimator's results change with other data samples (e.g., the test set) from same data generator.

$\rightarrow$ Regularization = attempt to combat the bias-variance tradeoff: to produce a $\tilde{\theta}_m$ with low bias **and** low variance.

Types of Regularization

- Parameter Norm Penalization
- Data Augmentation
- Multitask Learning
- Early Stopping
- Sparse Representations
- Ensemble Learning
- Dropout
- Adversarial Training

Parameter Norm Penalization

Some parameter sets, such as NN weights, achieve over-fitting when many parameters (weights) have a high absolute value. So incorporate this into the loss function.

$$\tilde{L}(\theta, C) = L(\theta, C) + \alpha\Omega(\theta)$$

L = loss function; $\tilde{L}$ = regularized loss function; C = cases; θ = parameters of the estimator (e.g. weights of the NN); Ω = penalty function; α = penalty scaling factor

L^2 Regularization

- $\Omega_2(\theta) = \frac{1}{2}\sum_{w_i \in \theta} w_i^2$
  Sum of squared weights across the whole neural network.

L^1 Regularization

- $\Omega_1(\theta) = \sum_{w_i \in \theta} |w_i|$
  Sum of absolute values of all weights across the whole neural network.

L_1 -vs- L_2 Regularization

Though very similar, they can have different effects.

- In cases where most weights have a small absolute value (as during the initial phases of typical training runs), L_1 imposes a much stiffer penalty: $|w_i| > w_i^2$.
- Hence, L_1 can have a sparsifying effect: it drives many weights to zero.
- Comparing penalty derivatives (which contribute to $\frac{\partial \tilde{L}}{\partial w_i}$ gradients):
  - $\frac{\partial \Omega_2(\theta)}{\partial w_i} = \frac{\partial}{\partial w_i}\left(\frac{1}{2}\sum_{w_k \in \theta} w_k^2\right) = w_i$
  - $\frac{\partial \Omega_1(\theta)}{\partial w_i} = \frac{\partial}{\partial w_i}\sum_{w_k \in \theta} |w_k| = sign(w_k) \in \{-1, 0, 1\}$
- So L_2 penalty scales linearly with w_i, giving a more stable update scheme. L_1 provides a more constant gradient that can be large compared to w_i (when $|w_i| < 1$).
- This also contributes to L_1's sparsifying effect upon θ.
- Sparsification supports feature selection in Machine Learning.

Dataset Augmentation

- It is easier to overfit small data sets. By training on larger sets, generalization naturally increases.
- But we may not have many cases.
- Sometimes we can **manufacture** additional cases.

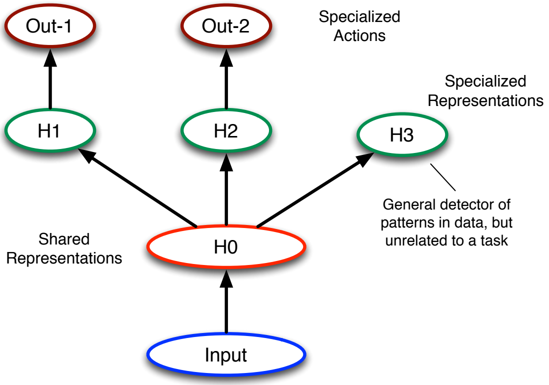
Approaches to Manufacturing New Data Cases

1. Modify the features in many (small or big) ways, but which we know will not change the target classification. E.g. rotate or translate an image.
2. Add (small) amounts of noise to the features and **assume** that this does not change the target. E.g., add noise to a vector of sensor readings, stock trends, customer behaviors, patient measurements, etc.

- Another problem is noisy data: the targets are wrong.
- Solution: Add (small) noise to all target vectors and use softmax on network outputs. This prevents overfitting to bad cases and thus improves generalization.

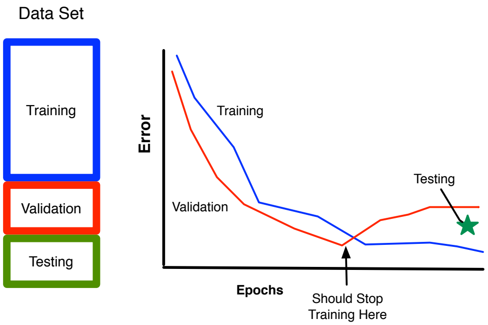
Multitask Learning

- Representations learned for task 1 can be reused for task 2, thus requiring fewer cases and less training to learn task 2.
- By forcing the net to perform multiple tasks, its shared hidden layer (H0) has general-purpose representations.


Out-1
Out-2
Specialized Actions
Specialized Representations
H1
H2
H3
General detector of patterns in data, but unrelated to a task
Shared Representations
H0
Input

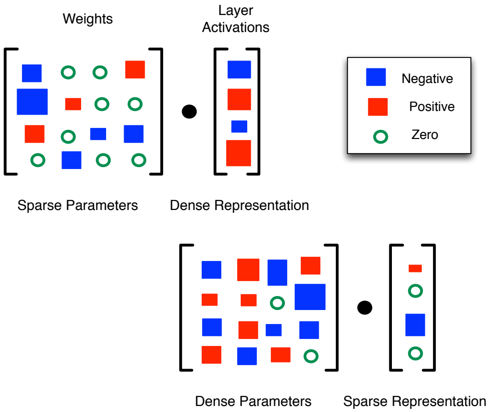
Early Stopping

- Divide data into training, validation and test sets.
- Check the error on the validation set every K epochs, but do **not** learn (i.e. modify weights) from these cases.
- Stop training when the validation error rises.
- Easy, non-obtrustive form of regularization: no need to change the network, the loss function, etc. Very common.



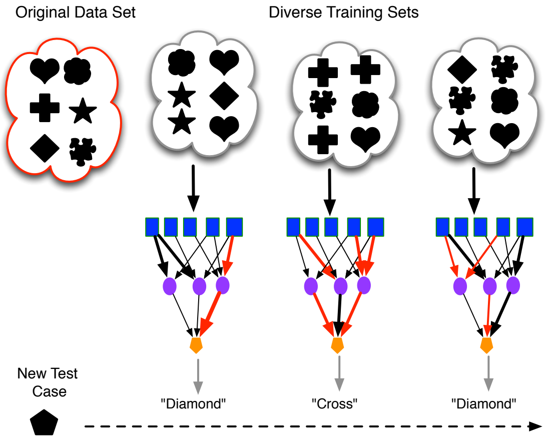
Sparse Representations

- Instead of penalizing parameters (i.e., weights) in the loss function, penalize hidden-layer activations.
- $\Omega(H)$ where H = hidden layer(s) and Ω can be L_1, L_2 or others.
- Sparser reps $\rightarrow$ better pattern separation among classes $\rightarrow$ better generalization.

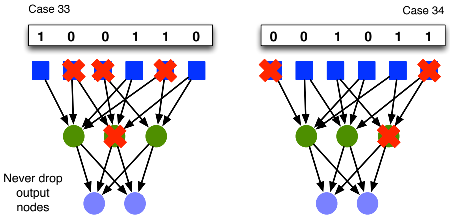


Ensemble Learning

- Train multiple classifiers on different versions of the data set.
- Each has different, but **uncorrelated** (key point) errors.
- Use voting to classify test cases.
- Each classifier has high variance, but the ensemble does not.

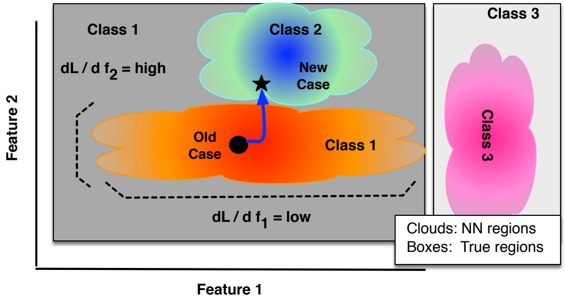


Dropout



- For each case, randomly select nodes at all levels (except output) whose outputs will be clamped to 0.
- Dropout prob range: (0.2, 0.5) - lower for input than hidden layers.
- Similar to bagging, with each model = a submodel of entire NN.
- Due to missing non-output nodes (but fixed-length outputs and targets) during training, each connection has more significance and thus needs (and achieves by learning) a higher magnitude.
- Testing involves **all** nodes.
- Scaling: prior to testing, multiply all weights by (1- p_h), where p_h = prob. dropping hidden nodes.

Adversarial Training



- Create new cases by mutating old cases in ways that maximize the chance that the net will misclassify the new case.
- Calc $\frac{\partial L}{\partial f_i}$ for loss func (L) and each input feature f_i.
- $\forall i : f_i \leftarrow f_i + \lambda \frac{\partial L}{\partial f_i}$ - make largest moves along dimensions most likely to increase loss/error; λ is very small positive.
- Train on the new cases $\rightarrow$ Increased generality.

Optimization

- Classic Optimization Goal: Reduce training error
- Classic Machine Learning Goal: Produce a **general** classifier.

  $\rightarrow$ Low error on training **and** test set.
  $\rightarrow$ Finding the **global** minimum in the error (loss) landscape is less important; a **local** minimum may even be a better basis for good performance on the test set.

Techniques to Improve Backprop Training

- Momentum
- Adaptive Learning Rates
- Batch Normalization
- Higher Order Derivatives

Smart Search

- Optimization $\rightarrow$ Making smart moves in the K+1 dimensional error (loss) landscape; K = $\|$ parameters $\|$ (wgts + biases).
- Ideally, this allows fast mvmt to global error mimimum.

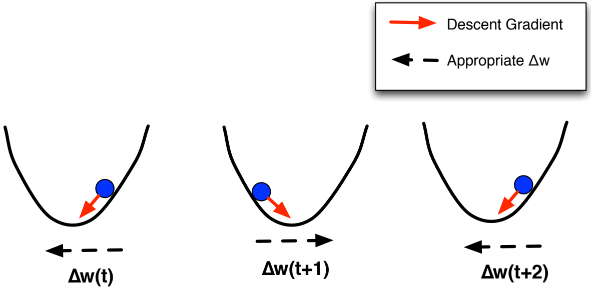
$$\triangle \mathbf{w_i} = -\lambda \frac{\partial(Loss)}{\partial w_i}$$

Techniques for Smart Movement

λ : Make step size appropriate for the texture (smoothness, bumpiness) of the current region of the landscape.

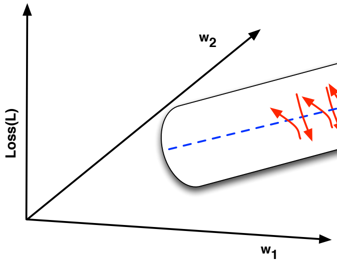
$\frac{\partial(Loss)}{\partial w_i}$: Avoid domination by the current (local) gradient by including previous gradients in the calculation of $\triangle w_i$.

Overshoot



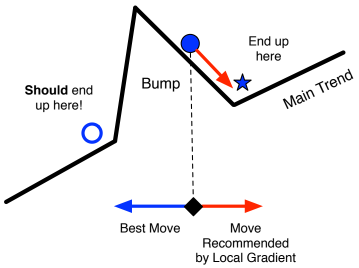
- When current search state is in a bowl (canyon) of the search landscape, following gradients can lead to oscillations if the step size is too big.
- Step size is λ (learning rate): $\triangle w = -\lambda \frac{\partial L}{\partial w}$.

Inefficient Oscillations



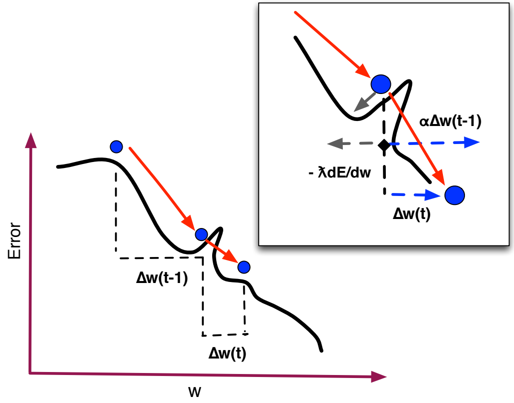
- Following the (weaker) gradient along w_1 leads to the minimum loss, but the w_2 gradients, up and down the sides of the cylinder (canyon), cause excess lateral motion.
- Since the w_1 and w_2 gradients are independent, search should still move downhill quickly (in this smooth landscape).
- But in a more rugged landscape, the lateral movement could visit locations where the w_1 gradient is untrue to the general trend. E.g. a little dent or bump on the side of the cylinder could have gradients indicating that **increasing** w_1 would help reduce Loss (L).

Dangers of Oscillation in a Rugged Landscape



How do we prevent local gradients from dominating the search?

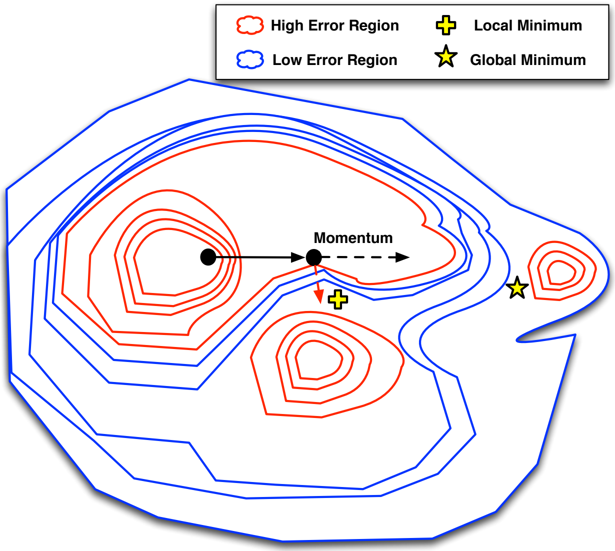
Momentum: Combatting Local Minima



$$\Delta w_{ij}(t) = -\lambda \frac{\partial E}{\partial w_{ij}} + \alpha \Delta w_{ij}(t-1)$$

Without momentum, $\lambda \frac{\partial E}{\partial w}$ leads to a local minimum

Momentum in High Dimensions



Momentum in Stochastic Gradient Descent (SGD)

- λ = learning rate; θ = wgts + biases.
- α = momentum factor (with typical value = 0.5, 0.9, 0.99)

The main SGD Loop (with Standard Momentum)

- Calc gradients: $g_t \leftarrow \left| \frac{\partial L}{\partial \theta} \right|_{\theta_t}$
- Update velocity:$v_{t+1} \leftarrow \alpha v_t - \lambda g_t$
- Update weights and biases: $\theta_{t+1} \leftarrow \theta_t + v_{t+1}$

The main SGD Loop (with Nesterov Momentum)

- Calc gradients: $g \leftarrow \left| \frac{\partial L}{\partial \theta} \right|_{\theta_t + \alpha v_t}$
- Update velocity:$v_{t+1} \leftarrow \alpha v_t - \lambda g$
- Update weights and biases: $\theta_{t+1} \leftarrow \theta_t + v_{t+1}$

Key difference: evaluates gradients using previous velocity

Adaptive Learning Rates

These methods have one learning rate **per parameter** (i.e. weight and bias).

Batch Gradient Descent

- Running **all** training cases before updating any weights.
- Delta-bar-delta Algorithm (Jacobs, 1988)
  - z = any system parameter.
  - While $sign(\frac{\partial Loss}{\partial z})$ is constant, learning rate $\uparrow$.
  - When $sign(\frac{\partial Loss}{\partial z})$ changes, learning rate $\downarrow$

Stochastic Gradient Descent (SGD)

Running a **minibatch**, then updating weights.

- Scale each learning rate inversely by accumulated gradient **magnitudes** over time. So learning rates tend to decrease over time; the question is often: *How quickly*? Caveat: some of the gradient accumulators do a weighted average that may decrease over time.
- More high-magnitude (pos or neg) gradients $\rightarrow$ learning rate $\downarrow$ quicker.
- Popular variants: AdaGrad, RMSProp, Adam

Basic Framework for Adaptive SGD

1. λ = a tensor of learning rates, one per parameter.
2. Init each rate in λ to same value.
3. Init the global gradient accumulator (R) to 0.
4. Minibatch Training Loop
   - Sample a minibatch, M.
   - Reset local gradient accumulator: $G \leftarrow ZeroTensor$.
   - For each case (c) in M:
     - Run c through the network; compute loss and gradients (g).
     - G $\leftarrow$ G + g (for all parameters)
   - $R \leftarrow f_{accum}(R, G)$
   - $\Delta\theta = -1 \times f_{scale}(R, \lambda) \odot G$
     where $\theta =$ weights and biases, and $\odot =$ element-wise operator: multiply each gradient by its own scale factor.

Key differences between optimizers: f_{accum} and f_{scale}

Specializations

AdaGrad (Duchi et. al., 2011)

- $f_{accum}(R,G) = R + G \odot G$ (square each gradient in G)
- $f_{scale}(R,\lambda) = \frac{\lambda}{\delta + \sqrt{R}}$ (where δ is very small, e.g. 10^{-7})

RMSProp (Hinton, 2012)

- $f_{accum}(R,G) = \rho R + (1-\rho) G \odot G$ (where $\rho =$ decay rate)
- $f_{scale}(R,\lambda) = \frac{\lambda}{\sqrt{\delta + R}}$ (where δ is very small, e.g. 10^{-6})

  ρ controls weight given to older gradients.

Adam (adaptive moments) (Kingma and Ba, 2014)

- $f_{accum}(R,G) = \frac{\rho R + (1-\rho) G \odot G}{1-\rho^T}$ (where $\rho =$ decay rate, T = timestep)
- $f_{scale}(R,\lambda) = \frac{\lambda S}{\delta + \sqrt{R}}$ ($\delta = 10^{-8}$)

  where $S \leftarrow \frac{\phi S + (1-\phi) G}{1-\phi^T}$ (S = first-moment estimate, ϕ = decay)

RMSProp and Adam are the most popular.

Momentum in Adam

- Adagrad and RMSprop use momentum only indirectly via the *2nd order moment estimate* (R).
- The S term in the Adam optimizer is a *1st-order moment estimate*: a more direct use of previous gradients to affect $\Delta\theta$.
- It implements momentum, to produce a modified value of the gradient G.
- Since S (and thus G) is included in f_{scale}, we do not need G in the final calculation of $\triangle\theta$, which, in Adam, is:

$$\triangle\theta = -1 \times f_{scale}(R, \lambda)$$

Batch Normalization

Normalize all activations in a layer w.r.t. minibatch averages.

Training Phase

- Hidden layer (h) of size n; Minibatch of size m.
- h_{ik} = output of kth neuron for case i of the minibatch
- Calculate averages and standard deviations (per neuron):

$$\mu_k = \frac{1}{m} \sum_{i=1}^{m} h_{ik}$$

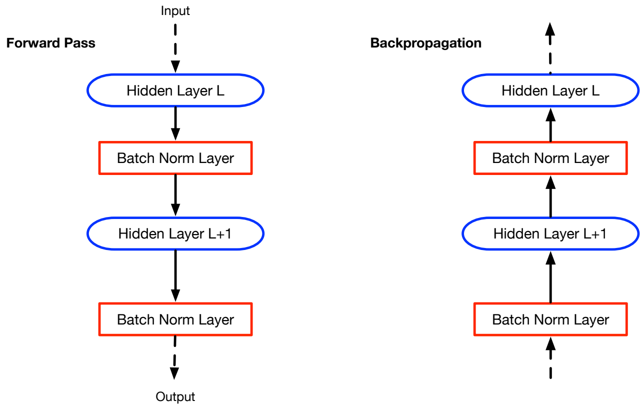
$$\sigma_k = \sqrt{\delta + \frac{1}{m} \sum_{i=1}^{m} [h_{ik} - \mu_k]^2}$$

- δ = small constant (e.g. 10^{-7}) to avoid divide by 0.
- Scale activations, $h_{ik}\ \forall i, k$:

$$\hat{h}_{ik} = \frac{h_{ik} - \mu_k}{\sigma_k}$$

Testing Phase: Calc $\hat{h}_{ik}$ using μ_k and σ_k from ALL training data.

Backpropagation with Batch Normalization (BN)



- Gradients $\frac{\partial Loss}{\partial w}$ easily calculated across batch norm layers.
- BN can be used either a) after F_{act} or b) between Σ inputs and F_{act}.

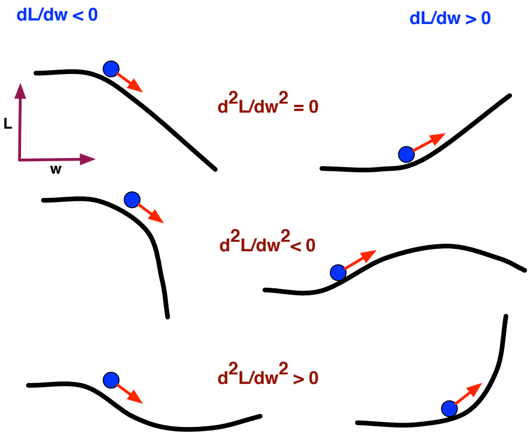
Why Is BN A Popular Optimizer?

- Normalization $\rightarrow$ many small outputs $\rightarrow$ less saturation of output functions $\rightarrow$ fewer vanishing gradients.
- Although sigmoid and tanh also have small outputs, the BN layer maintains significant gradients.
- Higher learning rates can be used with BN.
- Even sigmoids and tanhs used in hidden layers with BN.
- **BN Handles a Covariate Shift**
  - Covariate Shift $\rightarrow$ Distribution of features differs between training and test data.
  - Problem: Train on one ”type” of inputs, test on another. E.g. pictures of dogs under different lighting.
  - Testing goes poorly unless training and test data both scaled to same distribution.
  - Each hidden layer receives ”features” from the previous layer. It’s harder to learn when those features are always changing due to a) different data, and b) learning (which changes upstream weights and outputs).

Why Is BN a Useful Regularizer?

- In scaling the inputs and other activations, the use of minibatch statistics (avgs and variances) instead of those for the full batch introduces **noise** into the data (and processing).
- This prevents over-fitting in ways similar to both dataset augmentation and dropout.

Second Derivatives: Quantifying Curvature



- When $sign(\frac{\partial L}{\partial w}) = sign(\frac{\partial^2 L}{\partial w^2})$, the current slope gets more extreme.
- For gradient **descent** learning: $\frac{\partial^2 L}{\partial w^2} < 0$ (> 0) $\rightarrow$ More (Less) descent per $\triangle w$ than estimated by the standard gradient, $\frac{\partial L}{\partial w}$.

Approximations using 1st and 2nd Derivatives

- For a point (p_0) in search space for which we know F(p_0), use 1st and 2nd derivative of F at p_0 to estimate F(p) for a nearby point p.
- Knowing F(p) affects decision of moving to (or toward) p from p_0.
- Using the 2nd-order Taylor Expansion of F to approximate F(p):

$$F(p) \approx F(p_0) + F'(p_0)(p - p_0) + \frac{1}{2} F''(p_0)(p - p_0)^2$$

- Extend this to a multivariable function such as L (the loss function), that computes a scalar L(θ) when give the tensor of variables, θ:

$$L(\theta) \approx L(\theta_0) + (\theta - \theta_0)^T \left(\frac{\partial L}{\partial \theta} \right)_{\theta_0} + \frac{1}{2} (\theta - \theta_0)^T \left(\frac{\partial^2 L}{\partial \theta^2} \right)_{\theta_0} (\theta - \theta_0)$$

- $\frac{\partial L}{\partial \theta}$ = Jacobian, and $\frac{\partial^2 L}{\partial \theta^2}$ = Hessian
- Knowing L(θ) affects decision of moving to (or toward) θ from θ_0.

The Hessian Matrix

- A matrix (H) of all second derivatives of a function, f, with respect to all parameters.
- For gradient descent learning, f = the loss function (L) and the parameters are weights (and biases).

$$\mathbf{H}(L)(w)_{ij} = \frac{\partial^2 L}{\partial w_i \partial w_j}$$

- Wherever the second derivs are continuous, H is symmetric:

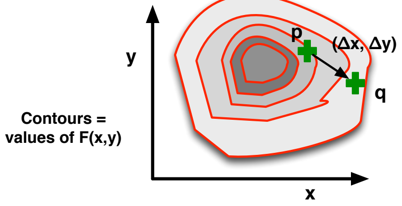
$$\frac{\partial^2 L}{\partial w_i \partial w_j} = \frac{\partial^2 L}{\partial w_j \partial w_i}$$

- Verify this symmetry: Let $L(x,y) = 4x^2y + 3y^3x^2$, and then compute $\frac{\partial^2 L}{\partial x \partial y}$ and $\frac{\partial^2 L}{\partial y \partial x}$

The Hessian Matrix

$$\mathbf{H}(L)(w)_{ij} = \begin{pmatrix} \frac{\partial L^2}{\partial w_1^2} & \frac{\partial L^2}{\partial w_1 \partial w_2} & \cdots & \frac{\partial L^2}{\partial w_1 \partial w_n} \\ \frac{\partial L^2}{\partial w_2 \partial w_1} & \frac{\partial L^2}{\partial w_2^2} & \cdots & \frac{\partial L^2}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial L^2}{\partial w_n \partial w_1} & \frac{\partial L^2}{\partial w_n \partial w_2} & \cdots & \frac{\partial L^2}{\partial w_n^2} \end{pmatrix}$$

Function Estimates via Jacobian and Hessian



- Given: point p, point q, F(p), the Jacobian(J) and Hessian(H) at p.
- Goal: Estimate F(q)
- Approach: Combine $\triangle p = [\triangle x, \triangle y]^T$ with J and H.

The Jacobian's Contribution to $\triangle F(x,y)$

$$[\triangle x, \triangle y] \bullet \left| \begin{matrix} \frac{\partial F}{\partial x} \\ \\ \frac{\partial F}{\partial y} \end{matrix} \right|_p \approx \triangle F(x,y)_{|_{p \to q}}$$

The Hessian's Contribution to $\triangle F(x,y)$

Hessian $\bullet \triangle p \approx$ Jacobian

$$\begin{bmatrix} \frac{\partial F^2}{\partial x^2} & \frac{\partial F^2}{\partial x \partial y} \\ \frac{\partial F^2}{\partial y \partial x} & \frac{\partial F^2}{\partial y^2} \end{bmatrix}_p \bullet \begin{vmatrix} \triangle x \\ \triangle y \end{vmatrix} \approx \begin{vmatrix} \frac{\partial F}{\partial x} \\ \frac{\partial F}{\partial y} \end{vmatrix}_p$$

$\triangle p^T \bullet Hessian \bullet \triangle p \approx \triangle F(x,y)$

$$[\triangle x, \triangle y] \bullet \begin{bmatrix} \frac{\partial F^2}{\partial x^2} & \frac{\partial F^2}{\partial x \partial y} \\ \frac{\partial F^2}{\partial y \partial x} & \frac{\partial F^2}{\partial y^2} \end{bmatrix}_p \bullet \begin{vmatrix} \triangle x \\ \triangle y \end{vmatrix} \approx$$

$$\approx [\triangle x, \triangle y] \bullet \begin{vmatrix} \frac{\partial F}{\partial x} \\ \frac{\partial F}{\partial y} \end{vmatrix}_p \approx \triangle F(x,y)_{|p \to q}$$

Eigenvalues of The Hessian Matrix

- For any unit vector, v, the 2nd derivative of L in that direction is $v^T Hv$.
- Since the Hessian is real and symmetric, it decomposes into an eigenvector basis and a set of real eigenvalues.
- For each eigenvector-eigenvalue pair (v_i, κ_i), where each v_i is a unit vector, the 2nd derivative of L in the direction v_i is κ_i, since:

$$v_i^T Hv_i = v_i^T \kappa_i v_i = v_i^T v_i \kappa_i = (1)\kappa_i$$

  - $Hv_i = \kappa_i v_i$ by definition of Eigenvectors and Eigenvalues.
  - $v_i^T v_i = 1$ since v_i is a unit vector.
- The max (min) eigenvalue = the max (min) second derivative along any of the eigenvectors. These indicate directions of high positive and negative curvature, along with flatter directions, all depending upon the signs and magnitudes of the eigenvalues.

The Condition Number of the Hessian

- Eigenvalues of the Hessian provide a quick check as to how dramatic and varied the curvatures of the loss function are at any point in parameter space.
- Condition number of a matrix = ratio of magnitudes of max and min eigenvalues, κ:

$$max_{i,j} \left| \frac{\kappa_i}{\kappa_j} \right|$$

- ILL Conditioning: High condition number (at a location in search space) $\rightarrow$ difficult gradient-descent search. Curvatures vary greatly in different directions, so the shared learning rate (which affects movement in all dimensions) may cause too big a step in some directions, and too small in others.

Optimal Step Sizes based on Hessian Eigenvalues

When derivative of loss function is negative along eigenvector v_i:

> $\kappa_i \approx 0 \rightarrow$ landscape has constant slope $\rightarrow$ take a normal-sized step along v_i, since the gradient is an accurate representation of the surrounding area.
>
> $\kappa_i > 0 \rightarrow$ landscape is curving upward $\rightarrow$ only take a small step along v_i, since a normal step could end up in a region of **increased** error/loss.
>
> $\kappa_i < 0 \rightarrow$ landscape is curving downward$\rightarrow$ take a large step along v_i to descend quickly.

The proximity of the ith dimensional axis to each such eigenvector indicates the proper step to take along dimension i: the proper $\triangle w_i$.