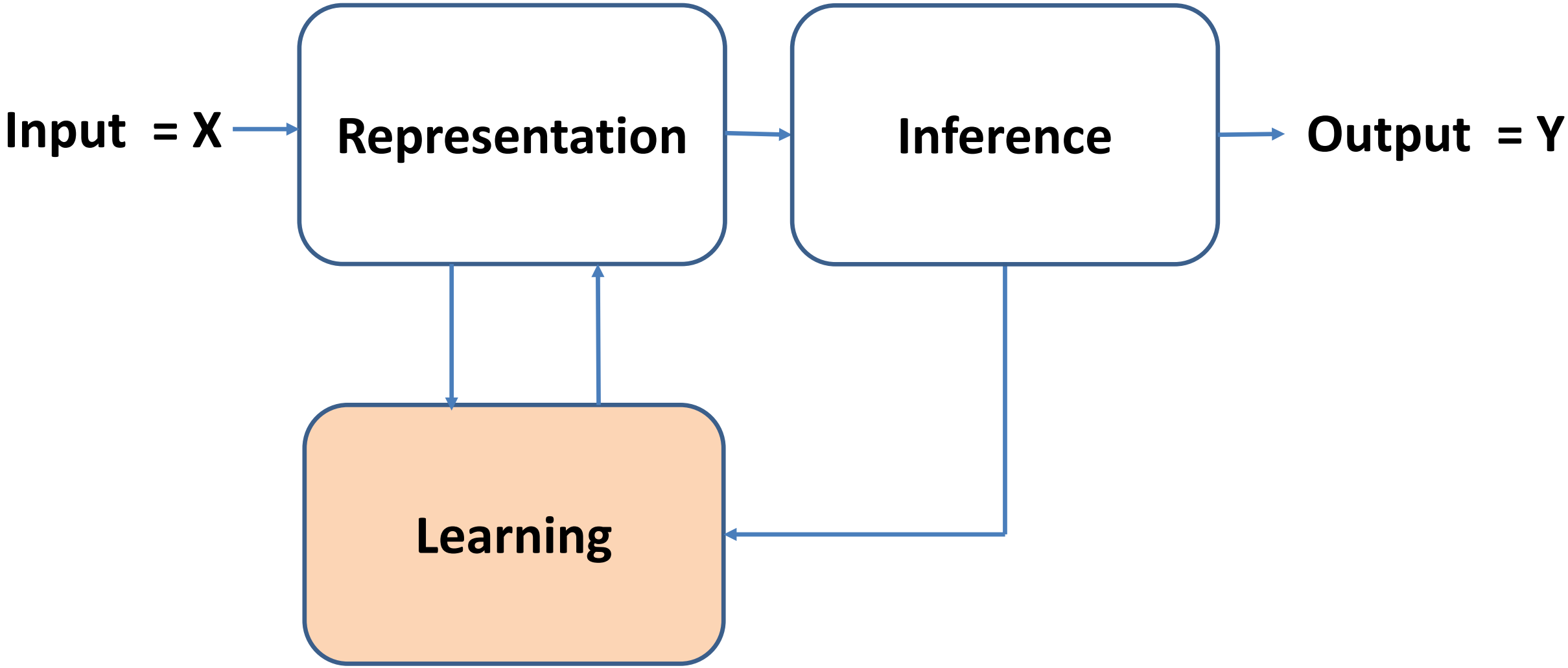
05 | Optimization for Deep Learning

Stephen F Elston | Principle Consultant, Quantia Analytics, LLC

Optimization for Deep Neural Networks



Optimization for Deep Neural Networks

- Neural networks **learn weights** using the backpropagation algorithm
- Weights are learned using the **gradient descent** method:

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

Where:

W_t = the tensor of weights or model parameters at step t

$J(W)$ = loss function given the weights

$\nabla_W J(W)$ = gradient of J with respect to the weights W

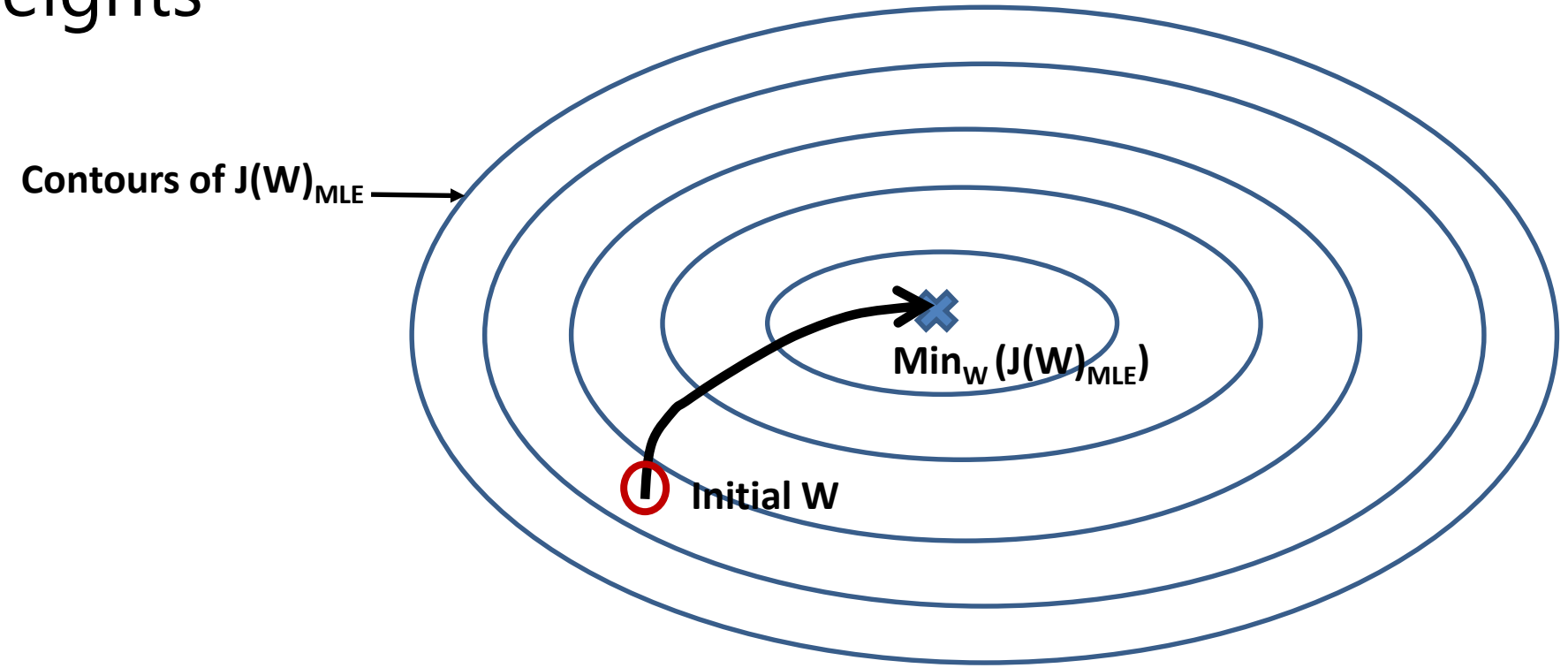
α = step size or learning rate

Optimization for Deep Neural Networks

- With millions of weights, we need highly efficient and reliable algorithms for gradient descent
- Overview of this module
  - Convergence problems
  - Batch gradient descent
  - Stochastic gradient descent
  - Adaptive stochastic gradient descent

Local Convergence of Gradient Descent

- Ideally, the loss function, J(W), is **convex** with respect to the weights



- Convex loss function has **one unique minimum**
- **Convergence** for convex loss function is **guaranteed**

Local Convergence of Gradient Descent

Expand loss function to understand convergence properties of gradient descent:

$$J(W^{(l+1)}) = J(W^{(l)}) + (W^{(l+1)} - W^{(i)})\vec{g} + \frac{1}{2}(W^{(l+1)} - W^{(i)})^T H(W^{(l+1)} - W^{(i)})$$

Where:

$W^{(l)}$ is the tensor of weights at step l

$\vec{g}$ is the gradient vector

H is the **Hessian** matrix.

Local Convergence of Gradient Descent

- How can you understand the Hessian matrix?

$$H\left(f(\vec{x})\right) = \begin{bmatrix} \frac{\partial^2 f(\vec{x})}{\partial x_1^2} & \frac{\partial^2 f(\vec{x})}{\partial x_2 \partial x_1} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_n \partial x_1} \\ \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_2} & \frac{\partial^2 f(\vec{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_n} \\ \vdots & \vdots & \vdots & \vdots \\ \frac{\partial^2 f(\vec{x})}{\partial x_1 \partial x_n} & \frac{\partial^2 f(\vec{x})}{\partial x_2 \partial x_n} & \cdots & \frac{\partial^2 f(\vec{x})}{\partial x_n^2} \end{bmatrix}$$

- The Hessian is the matrix of derivatives of the gradient
- Properties of the Hessian determine convergence rate

Local Convergence of Gradient Descent

- Let's look at a solution for the convex optimization problem:

$$J(W^{(l+1)}) = J(W^{(l)}) + (W^{(l+1)} - W^{(i)})\vec{g} + \frac{1}{2}(W^{(l+1)} - W^{(i)})^T H(W^{(l+1)} - W^{(i)})$$

- Given a step size α we can rewrite the above **quadradic equation**:

$$J(W^{(l)} - \alpha\vec{g}) = J(W^{(l)}) - \alpha\vec{g}^T\vec{g} + \frac{1}{2}\alpha^2\vec{g}^T H\vec{g}$$

- The minimum occurs where the **gradient is 0**:

$$J(W^{(l)}) = J(W^{(l)} - \alpha\vec{g})$$

- And, with **optimal step size**: $\alpha^* = \frac{\vec{g}^T\vec{g}}{\vec{g}^T H\vec{g}}$

Local Convergence of Gradient Descent

- Real-world loss functions are **typically not convex**
- There can be multiple minimums and maximums; a **multi-modal loss function**
- Finding the **globally optimal solution** is hard!
- The minimum reached by an optimizer depends on the **starting value of W**
- In practice, we are happy with a **good local solution**, if not, the globally optimal solution
- First order optimization found to perform as well, or better, than second order

The Nature of Gradients

- To understand the behavor of the Hessian we need to examine the **eigenvalues**
- A square matrix can be decomposed into eigenvalues and **eigenvectors**: $A = Q\Lambda Q^{-1}$
- Where Q is the matrix of **unitary** eigenvectors
- The eigenvalues are a diagonal matrix

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & 0 & \dots & 0 \\ 0 & \lambda_2 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & & \vdots \\ 0 & 0 & 0 & \dots & \lambda_m \end{bmatrix}$$

The Nature of Gradients

- Some key properties of the Hessian matrix:
  - The Hessian is symmetric since $\frac{\partial f(\vec{x})}{\partial x_1 \partial x_2} = \frac{\partial f(\vec{x})}{\partial x_2 \partial x_1}$
  - For a convex loss function the Hessian has all **positive eigenvalues**; it is **positive definite**
  - At a maximum point the Hessian has all **negative eigenvalues**; it is **negative definite**
  - The Hessian has **some positive and some negative eigenvalues** at a **saddle point**
  - Saddle points are problematic since direction of descent to the minimum is unclear
  - If Hessian has some **very small eigenvalues**, the gradient is low and **convergence will be slow**

The Nature of Gradients

- For quadratic optimization, the rate of convergence is determined by the **condition number** of the Hessian:

$$\kappa(H) = \frac{|\lambda_{max}(H)|}{|\lambda_{min}(H)|}$$

- Where:

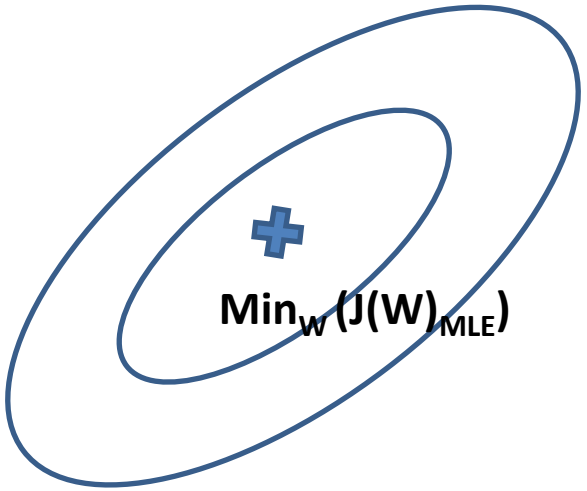
$|\lambda_{max}(H)|$ is the absolute value of the largest eigenvalue of H

$|\lambda_{min}(H)|$ is the absolute value of the smallest eigenvalue of H

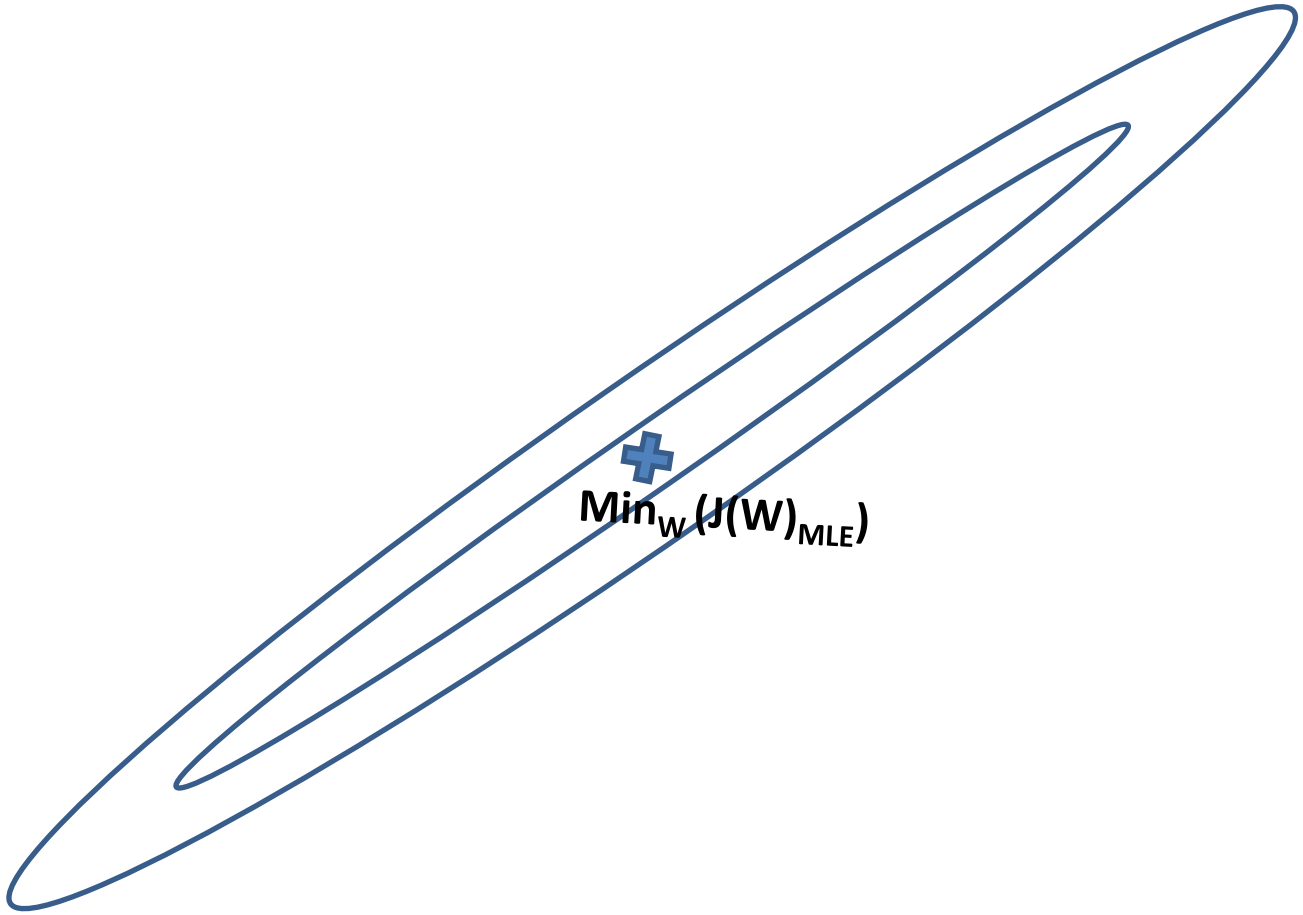
- If the condition number is close to 1.0, the Hessian is **well conditioned** and convergence will be fast
- If the condition number is large, the Hessian is **ill-conditioned** and convergence will be slow; gradient is flat in some dimensions

The Nature of Gradients

- Example of well-conditioned and ill-conditioned gradients:



well-conditioned gradient



ill-conditioned gradient

Vanishing and Exploding Gradient Problems

- There is no guarantee that the gradient of the loss function is well behaved
- The gradient can **vanish**
  - Flat spots in the gradient
  - Imagine a loss function with a long narrow valley
  - Ill-conditioned Hessian
  - Slow convergence
- The gradient can **explode**
  - Sudden changes in the gradient; falling off a cliff!
  - Ill-conditioned Hessian
  - Very large step; optimizer over-shoots the minimum point

Vanishing and Exploding Gradient Problems

- What happens to the eigenvalues of the Hessian?
- Consider an n layer NN with linear activation
  - The gradient is just the weight tensor, W, with eigen decomposition:

$$W = Q\Lambda Q^T$$

- Multiplying the n weight tensors for the multi-layer NN:

$$W^n = \left(Q\Lambda Q^T\right)^n = Q\Lambda^n Q^T$$

- If the eigenvalue << 1.0, the gradient vanishes
- If the eigenvalue >> 1.0, the gradient explodes

Vanishing and Exploding Gradient Problems

- What can be done about extreme gradient problems?
- Dealing with vanishing gradient can be difficult
  - Normalization of input values
  - Regularization can help
- Dealing with exploding gradients is easy
  - **Gradient clipping** prevents extreme values

Batch Gradient Descent

- Recall the basic gradient descent equation:

$$W_{t+1} = W_t + \alpha \nabla_W J(W_t)$$

- Where:

W_t = the tensor of weights or model parameters at step t

$\nabla_W J(W)$ = gradient of J with respect to the weights W

α = step size or learning rate

- The gradients of the multi-layer NN are computed using the chain rule

Batch Gradient Descent

- Can we use the gradient descent equation directly?
- Yes, we can
- Iterate the weight tensor relation until a **stopping criteria** or **error tolerance** is reached:

$$||W^{t+1} - W^{t}|| < \text{tolerance}$$

- But;
  - Must compute the gradient for all weights at one time as a **batch** at each step
  - Does not scale if there are a large number of weights

Stochastic Gradient Descent

- We need a more scalable way to apply gradient descent
- Stochastic gradient descent is just such a method
- The weight tensor update for stochastic gradient descent follows this relationship:

$$W_{t+1} = W_t + \alpha \, E_{\hat{p}data}\left[\nabla_W J(W_t)\right]$$

Where:

$\hat{p}data$ is the Bernoulli sampled mini-batch

$E_{\hat{p}data}[]$ is the expected value of the gradient given the Bernoulli sample

Stochastic Gradient Descent

- Stochastic gradient descent is known to converge well in practice
- Empirically, using mini-batch samples provide a better exploration of the loss function space
  - Can help solution escape from small local gradient problems
  - Sampling is dependent on mini-batch size

Stochastic Gradient Descent

- Stochastic gradient descent algorithm
  1. A randomly Bernoulli sort the samples
  2. Round-robin, take a mini-batch sample of the gradient
  3. The expected value of the gradient is computed
  4. The weight tensor is updated
  5. Repeat 2-4 above until stopping criteria is reached
- Notice that the addition rounds repeat the samples
  - In practice this does not create much bias
  - For large samples this may not happen

Stochastic Gradient Descent with Momentum

- The stochastic gradient descent algorithm can be slow to converge if flat spots in the gradient are encountered
- What is a solution?
- Add **momentum** to the gradient; $momentum = m \cdot v$
  - Analogy with Newtonian mechanics;
  - Where:

m is the mass
v is the velocity

Stochastic Gradient Descent with Momentum

Letting the mass be 1.0 update of the weight tensor is:

$$v^{(l)} = momentum \cdot v^{(l-1)} + lr \cdot \nabla_W J(W^{(l)})$$

$$W^{(l+1)} = W^{(l)} + v^{(l)}$$

Where:

$v^{(l)}$ is the velocity at step l
$momentum$ is the momentum multiplier
lr is the learning rate

Notice there are now two hyperparameters

Adaptive Stochastic Gradient Descent

- A single learning rate is not likely to be optimal
  - Far from the minimum, a large learning rate speeds convergence
  - Near the minimum a small learning rate presents over-shooting the minimum
- What can improve the convergence
- Use a manually created learning schedule
  - Introduces additional hyperparameters
- Use an adaptive algorithm
  - Learning rate is adjusted based on the estimates of the gradient

Selecting Initial Weight Values

- To prevent weights from becoming linearly dependent the **initial values must be randomly selected**
  - Otherwise, some weight values are never learned
- Simple truncated Gaussian or Uniform distributed values work well in practice
  - This process is referred as adding **fuzz** to the initial values

Microsoft