Gradient Descent Optimization
Gradient Descent vs Newton Optimization

\[ f = x^2 - y^2 \]
Newton Optimization

• Newton method in 1-D

\[ x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \]

• Multivariate Newton Method

\[ \mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H} f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n) \]
Multivariate Newton Method

\[ f = x^2 - y^2 \]

\[ \mathbf{x}_{n+1} = \mathbf{x}_n - [\mathbf{H} f(\mathbf{x}_n)]^{-1} \nabla f(\mathbf{x}_n) \]

\[
\mathbf{H} = \begin{bmatrix}
\frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\
\frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2}
\end{bmatrix}
\]

Invert it:

\[ [\mathbf{H} f]^{-1} = \begin{bmatrix} 1/2 & 0 \\ 0 & -1/2 \end{bmatrix} \]

Get the gradient:

\[ \nabla f = \begin{bmatrix} 2x \\ -2y \end{bmatrix} \]
SGD Algorithm

for i in range(0,10000):
    loss = 0
    for j in range(0,len(y)):
        a = w0*x0[j] + w1 * x1[j] + b  # forward pass
        loss += 0.5 * (y[j] - a)**2  # compute loss
        dw0 = -(y[j]-a)*x0[j]  # compute gradients
        dw1 = -(y[j]-a)*x1[j]
        db = -(y[j]-a)
        w0 = w0 - 0.001 * dw0  # update weights, biases
        w1 = w1 - 0.001 * dw1
        b = b - 0.001 * db
    print('loss =',loss)

\[ \theta = \theta - \eta \cdot \nabla_{\theta} J(\theta; x^{(i)}; y^{(i)}). \]
MiniBatch SGD Algorithm

```python
batch_size = 5
for i in range(0,50000):
    loss = 0
    for j in range(0,len(y)//batch_size):
        dw0, dw1, db = 0, 0, 0  # reset gradients in the batch
        for k in range(0,batch_size):
            index = j*batch_size+k
            a = w0*x0[index] + w1 * x1[index] + b  # forward pass
            loss += 0.5 * (y[index] - a)**2  # compute loss
            dw0 += -(y[index]-a)*x0[index]  # aggregate gradients
            dw1 += -(y[index]-a)*x1[index]
            db += -(y[index]-a)
        w0 = w0 - 0.001 * dw0/batch_size  # update weights,biases after
        w1 = w1 - 0.001 * dw1/batch_size  # accumulating gradients in a batch
        b = b - 0.001 * db/batch_size
    print('loss =',loss)
```

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$.
Gradient Descent Algorithm

• Deep Learning is a high dimensional optimization problem

• Saddle points are common in high dimensional data

• Newton method is not suitable for high dimensional optimization problems.

• Stochastic Gradient Descent *can* break out of simple saddle points, as updates are done in each dimension, and if the step size is large enough for it to go over the flatness.
Gradient Descent

• With random initialization, gradient descent will escape the saddle point. It has been shown that gradient descent without perturbation can take exponential time to escape saddle points (1703.00887.pdf (arxiv.org)).

• Deep neural networks have many saddle points so it can significantly slow down the training.

---

Algorithm 1 Perturbed Gradient Descent (Meta-algorithm)

\[
\text{for } t = 0, 1, \ldots \text{ do} \\
\quad \text{if perturbation condition holds then} \\
\quad \quad x_t \leftarrow x_t + \xi_t, \quad \xi_t \text{ uniformly } \sim B_0(r) \\
\quad \quad x_{t+1} \leftarrow x_t - \eta \nabla f(x_t) 
\]
Deep Network training via Minibatch SGD

• The search space of deep networks is highly complex with saddle points and local minima.

• In many cases, the global minima is close to the local minima.

• **Goal**: Find a good local minima as efficiently as possible.
Further Optimizing the SGD Algorithm

- Momentum

\[ v_t = \gamma v_{t-1} + \eta \nabla \theta J(\theta) \]
\[ \theta = \theta - v_t \]

Stochastic Gradient Descent (SGD)
Momentum

\[ v_t = \gamma v_{t-1} + \eta \nabla_{\theta} J(\theta) \]

\[ \theta = \theta - v_t \]
Nestrov Accelerated Gradient

\[ \nu_t = \gamma \nu_{t-1} + \eta \nabla_{\theta} J(\theta - \gamma \nu_{t-1}) \]
\[ \theta = \theta - \nu_t \]

We look ahead by calculating the gradient w.r.t. the approximate future position of our parameters
AdaGrad

- Adapt our updates to each individual parameters (weights/biases) to perform larger or smaller updates depending on their importance.

- Terminology: $g_{t,i} = \text{gradient of a parameter } i \text{ in step } t$.

- SGD
  \[
  \theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i}.
  \]

- AdaGrad ($G_t = \text{sum of square of gradients up to time step } t$)
  \[
  \theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}.
  \]

- Matrix form:
  \[
  \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t.
  \]

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element $i$, $i$ is the sum of the squares of the gradients w.r.t. $\theta_i$ up to time step $t$ \cite{12}, while $\epsilon$ is a smoothing term that avoids division by zero (usually on the order of $1e-8$).
AdaDelta

\[ \theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}. \]

- Adagrad's main weakness is its accumulation of the squared gradients in the denominator.
- Since every added term is positive, the accumulated sum keeps growing during training.
- This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge.
- *AdaDelta* uses a running time window to keep the square of the past gradients
AdaDelta

- The sum of gradients is recursively defined as a decaying average of all past squared gradients.
  \[ E[g^2]_t = \gamma E[g^2]_{t-1} + (1 - \gamma) g_t^2. \]

- SGD
  \[ \Delta \theta_t = -\eta \cdot g_{t,i} \]
  \[ \theta_{t+1} = \theta_t + \Delta \theta_t \]

- AdaGrad
  \[ \Delta \theta_t = -\frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t. \]

- AdaDelta
  \[ \Delta \theta_t = -\frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t. \]
  \[ \Delta \theta_t = -\frac{\eta}{RMS[g]_t} g_t. \]
RMSProp

• Similar to AdaDelta but independently developed.

\[
E[g^2]_t = 0.9E[g^2]_{t-1} + 0.1g_t^2
\]

\[
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} g_t
\]
Adam

- Adaptive Momentum Estimation

- Compute the decaying averages of past and past squared gradients $m_t$ (mean) and $v_t$ (variance) as:

  \[
  m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
  v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2
  \]

  \[
  \hat{m}_t = \frac{m_t}{1 - \beta_1^t} \\
  \hat{v}_t = \frac{v_t}{1 - \beta_2^t}
  \]

  \[
  \theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t.
  \]