- Recall our general iterative minimisation algorithm:
  
  ```python
  x = x0
  for k in range(num_iters):
    step = calcStep(fn, x)
    x = x - step
  ```

  and one way to choose the step, namely:

  $$\text{step} = \alpha [\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)]$$

- We’ve spent a good deal of time looking at how to select step-size/learning rate $\alpha$
- Now let’s go back and look at how we calculate $\frac{\partial f}{\partial x_1}(x)$ etc
Approximate Derivatives

\* \( f(x) \approx f(x') + \frac{\partial f}{\partial x_1}(x')(x_1 - x'_1) + \frac{\partial f}{\partial x_2}(x')(x_2 - x'_2) + \cdots + \frac{\partial f}{\partial x_n}(x')(x_n - x'_n) \)

\* Choosing \( x \) with \( x_1 = x' - \alpha \frac{\partial f}{\partial x_1}x' \), \( x_2 = x' - \alpha \frac{\partial f}{\partial x_2}x' \), \ldots , \( x_n = x' - \alpha \frac{\partial f}{\partial x_n}x' \) then

\[ f(x) \approx f(x') - \alpha \frac{\partial f}{\partial x_1}(x')^2 - \alpha \frac{\partial f}{\partial x_2}(x')^2 - \cdots - \alpha \frac{\partial f}{\partial x_n}(x')^2 \]

i.e. moving from point \( x' \) to \( x \) tends to decrease function \( f(\cdot) \)

\* This is already an approximation, so why not also use an approximation to the derivative i.e.

\[ x_1 = x' - \alpha Df_{x_1}(x') \]

with \( Df_{x_1}(x') \approx \frac{\partial f}{\partial x_1}(x') \), etc
Approximate Derivatives

* Let’s swap from $\min_x f(x)$ to $\min_\theta J(\theta)$ to be consistent with our ML notation. Functions we want to minimise in ML are often of the form:

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \text{loss}(\theta, x^{(i)}, y^{(i)})$$

where

* $(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots, (x^{(m)}, y^{(m)})$ is our training data
* $\text{loss()}$ is a function that measures how well our predictions match the training data
  
  e.g. $\text{loss}(\theta, x^{(i)}, y^{(i)}) = (\theta^T x^{(i)} - y^{(i)})^2$ in linear regression
* Derivatives:

$$\frac{\partial J}{\partial \theta_1}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \text{loss}}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)}), \quad \frac{\partial J}{\partial \theta_2}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \frac{\partial \text{loss}}{\partial \theta_2}(\theta, x^{(i)}, y^{(i)}) \text{ etc}$$

* Pick a random sample of $b$ points from the training data. Let $N$ be the set of indices. Then use approx derivatives

$$DJ_{\theta_1}(\theta) = \frac{1}{b} \sum_{i \in N} \frac{\partial \text{loss}}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)}), \quad DJ_{\theta_2}(\theta) = \frac{1}{b} \sum_{i \in N} \frac{\partial \text{loss}}{\partial \theta_2}(\theta, x^{(i)}, y^{(i)}) \text{ etc}$$

* When $b = m$ and $N = \{1, 2, \ldots, m\}$ then we get back exact derivatives.
Approximate Derivatives: Example

* Suppose we have \( m = 100 \) training data points \((x^{(i)}, y^{(i)})\), \(i = 1, 2, \ldots, 100\). Linear regression:

\[
loss(\theta, x^{(i)}, y^{(i)}) = 0.5(\theta^T x^{(i)} - y^{(i)})^2, \quad \frac{\partial loss}{\partial \theta_k}(\theta, x^{(i)}, y^{(i)}) = (\theta^T x^{(i)} - y^{(i)})x_k^{(i)}
\]

\[
J(\theta) = \frac{1}{100} \sum_{i=1}^{100} loss(\theta, x^{(i)}, y^{(i)}), \quad \frac{\partial J}{\partial \theta_k}(\theta) = \frac{1}{100} \sum_{i=1}^{100} \frac{\partial loss}{\partial \theta_k}(\theta, x^{(i)}, y^{(i)})
\]

* Sample \( b = 5 \) points randomly from training data e.g. \( N = \{76, 75, 40, 66, 18\} \), approx derivative:

\[
DJ_{\theta_1}(\theta) = \frac{1}{5} \sum_{i \in N} \frac{\partial loss}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)}) = \frac{1}{5} \sum_{i \in N} (\theta^T x^{(i)} - y^{(i)})x_1^{(i)}
\]

and so on for \( \theta_2, \ldots, \theta_n \)
Approximate Derivatives: Example

* Example: \( f = x^2 \), training data is 100 points with \( x^{(i)} \) random and \( y^{(i)} = (x^{(i)})^2 + \text{noise} \).

* Draw 5 random training data points and calc approx derivative. Repeat 5 times (so get 5 different approximations to the derivative):

![Graphs showing approximate derivatives with solid lines indicating exact derivatives.]

* Now use 50 random training data points to calc approx derivative:

![Graphs showing approximate derivatives with solid lines indicating exact derivatives.]

Observe the approximate derivatives tend to be more accurate now.
In above example we sampled 5 points with replacement from the 100 training data points. But this might mean that we don’t use some of our training data, even after drawing repeated samples.

Periodic sampling is more common:

- Shuffle (i.e. randomly permute) training data
- For first sample take points 1,2,...,5 from data
- For second sample take points 6,7,...,10 from data
- and so on
- After doing this $n/5$ times we will have used all of our training data
Mini-Batch Stochastic Gradient Descent (SGD)

* Modifying gradient descent to use approx derivatives and periodic sampling:

\[ \theta = \theta_0, \quad \text{batch\_size} = 5, \quad n = \#\text{training data points} \]

for \( k \) in range(num\_iters):
    shuffle training data
    for \( i \) in np.arange(0, \( n \), batch\_size):
        sample \( N=\text{np.arange}(i, i + \text{batch\_size}) \)
        calc approx derivative \( DJ(\theta) = [DJ_{\theta_0}(\theta), DJ_{\theta_1}(\theta), \ldots, DJ_{\theta_n}(\theta)] \)
        \[ \theta = \theta - \alpha DJ(\theta) \]

* Each run through the inner for-loop is called an epoch (corresponding to one run through the full training data).

* Why do this? When \#training data points \( n \) is very large, calculating approx derivative from small sample can be much faster than calculating exact derivative, yet still good enough to find downhill direction.

* Mini-batch SGD is the standard approach with neural nets
Example

* Quadratic loss $m = 1000$ training data points, $y = \theta^T x + \text{noise}$ with $\theta = [3, 4]$, starting point $x = [1, 1]$, constant $\alpha_0 = 0.5$

* Mini-batch SGD, batch size 5

* For SGD one epoch is one run through training data $\rightarrow x$ has been updated $m/batch\_size$ times. For GD one iteration is one update of $x$.

* With SGD get fast initial convergence, but when close to minimum the “noise” in the approx derivative causes $x$ to “wander”
**Example**

- Mini-batch SGD, batch size 5

  ![Graphs](image)

- Iteration here = an update to $x$, so one run of inner mini-batch loop
- Epoch = one run through all the mini-batches, so one run of outer loop.
- Epoch = $m/batch\_size = 1000/5 = 200$ iterations, so 50 epochs = 10000 iterations
* Mini-batch SGD, batch size 5

* Mini-batch SGD, batch size 25

* Increasing batch size reduces the “noise” in the approx derivative and reduces wandering by $x$ when close to minimum.
* Also slows fast initial convergence?
Example

* Quadratic loss $m = 1000$ training data points, $y = \theta^T x + \text{noise}$ with $\theta = [3, 4]$, starting point $x = [1, 1]$, constant $\alpha_0 = 0.5$

* Output $y$ noise std deviation 1, mini-batch SGD, batch size 5

* Output $y$ noise std deviation 5, mini-batch SGD, batch size 5

* More measurement noise causes mini-batch derivative estimate to get less accurate, $x$ wanders a lot more
Toy neural net \( m = 100 \) training data points, output \( y \) noise with std dev 0.05, starting point \( x = [1, 1] \), constant \( \alpha_0 = 0.75 \)

Mini-batch SGD, batch size 5

Faster initial convergence with SGD
Why Does Initial Convergence Seem Faster With SGD?

* Pick a random mini-batch sample from the training data. Let $N$ be the indices. Then use approx derivatives:

$$DJ_{\theta_1}(\theta; N) = \sum_{i \in N} \frac{\partial \text{loss}}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)}), \quad DJ_{\theta_2}(\theta) = \sum_{i \in N} \frac{\partial \text{loss}}{\partial \theta_2}(\theta, x^{(i)}, y^{(i)}) \text{ etc}$$

* When $N = \{1, 2, \ldots, m\}$ then we get back exact derivatives.

* Suppose we repeatedly draw random samples, calc $DJ_{\theta_1}(\theta)$ and plot the values. E.g. Quadratic loss and $y = \theta^T x + \text{noise}$, $\theta = [3, 4]$:

* The approximate derivatives $DJ_{\theta_i}$ take values around the exact derivatives $\frac{\partial J}{\partial \theta_i}, \ i = 0, 1$. Why?
Why Does Initial Convergence Seem Faster With SGD?

Think about drawing balls from a bag:

* Suppose I have a bag with \( m \) different balls in it.
* I reach into the bag to draw out a ball, each ball has probability \( \frac{1}{m} \) of being picked.
* After picking a ball, I put it back in the bag i.e. *sample with replacement*
* Repeat this \( b \) times, so selecting a batch of \( b \) balls.
* Each ball in the bag will on average appear \( \frac{b}{m} \) times in the batch, e.g. \( m = 100, \ b = 5 \) and repeat experiment \( M = 1000 \) times:

```
m=100; b=5; M=1000
count=np.zeros(m)
for k in range(M):
    batch=[]
    for i in range(b):
        batch.append(np.random.randint(0,m))
    count[batch] = count[batch]+1
plt.bar(range(m),count/M)
```
Why Does Initial Convergence Seem Faster With SGD?

* Pick a random mini-batch of $b$ samples from the training data, repeat $M$ times and let $N_k$ be the indices of the $k$’th sample. The $k$’th approx derivative is:

$$DJ_{\theta_1}(\theta; N_k) = \frac{1}{b} \sum_{i \in N_k} \frac{\partial \text{loss}}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)})$$

* There are $m$ training data points. Each point in the training data is selected to be in sample $N_k$ with probability $b/m$ (cf bag with $m$ balls from which we draw batches of size $b$).

* Over $M$ runs training point $i$ will on average be selected $Mb/m$ times i.e.

$$\frac{1}{M} \sum_{k=1}^{M} \frac{1}{b} \sum_{i \in N_k} \frac{\partial \text{loss}}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)}) \approx \frac{Mb}{Mb} \sum_{i=1}^{m} \frac{\partial \text{loss}}{\partial \theta_1}(\theta, x^{(i)}, y^{(i)}) = \frac{\partial J}{\partial \theta_1}(\theta)$$

* So the way that we sample mini-batches ensures that the empirical mean of the approx derivative equals to the exact derivative.

* Can think of $DJ_{\theta_1}(\theta; N_k) = \frac{\partial J}{\partial \theta_1}(\theta) + \text{noise}_k$, with $E[\text{noise}_k] = 0$

* i.e. $\frac{1}{M} \sum_{k=1}^{M} DJ_{\theta_1}(\theta; N_k) = \frac{\partial J}{\partial \theta_1}(\theta) + \frac{1}{M} \sum_{k=1}^{M} \text{noise}_k \to \frac{\partial J}{\partial \theta_1}(\theta)$ as $M \to \infty$
Why Does Initial Convergence Seem Faster With SGD?

* Quadratic loss $m = 100$ training data points, $y = \theta^T x + \text{noise}$ with $\theta = [3, 4]$, starting point $x = [1, 1]$, constant $\alpha_0 = 0.5$

* SGD plot: solid line is empirical mean over the $M$ runs, error bars indicate one standard deviation, $x$-axis is mini-batch inner loop iterations, *not epochs*.  

* See that on average the convergence rate of SGD matches that of gradient descent when we compare iterations (rather than epochs)

* One iteration of SGD is much cheaper than one iteration of gradient descent
Why Does Initial Convergence Seem Faster With SGD?

- Toy neural net $m = 100$ training data points, output $y$ noise with std dev 0.05, starting point $x = [1, 1]$, constant $\alpha_0 = 0.75$

Gradient descent

Mini-batch SGD, batch size 5. Repeat $M = 25$ times
In above analysis we selected each training point *uniformly at random with replacement* and treat mini-batches *independently*.

- Uniformly at random = every training point equally likely to be picked for inclusion in a mini-batch.
- With replacement = we put ball back in bag after picking it, so there’s a chance the same ball will be picked again i.e. a mini-batch may contain the same training point multiple times.
- Treat mini-batches independently = we repeat this process independently for each mini-batch, so the same training data point can also appear in two consecutive mini-batches.
- E.g. with $m = 10$ data points and mini-batch size 5 we might choose mini-batches:

$$\{5, 3, 5, 1, 2\}, \{1, 4, 6, 2, 5\}, \{3, 2, 6, 4, 4\}$$
**Sampling Strategy**

- Alternative: selected each training point *uniformly at random without replacement* and treat mini-batches *independently*.
  - Uniformly at random = every training point equally likely to be picked for inclusion in a mini-batch.
  - Without replacement = we *don’t* put ball back in bag after picking it, so a training point can only appear once in a mini-batch.
  - When number of training data points is large (as it usually is when using SGD) then with/without replacement strategies are much the same (the chance of choosing the same training point multiple times in a mini-batch is really small).
  - Treat mini-batches independently = we repeat this independently for each mini-batch, so the same training data point can also appear in two consecutive mini-batches even though the same point cannot appear twice within the same mini-batch.
  - E.g. with $m=10$ data points and mini-batch size $5$ we might choose mini-batches:

$$\{5, 3, 1, 2, 4\}, \{1, 4, 6, 2, 5\}, \{3, 2, 6, 1, 4\}$$
Alternative (more common): select each training point once over an epoch, but *in random order*. E.g. it’s what this code does:

\[
\begin{align*}
\theta &= \theta_0, \quad \text{batch}_\text{size} = 5, \quad n = \# \text{training data points} \\
\text{for } k \text{ in range}(\text{num}_\text{iters}): \\
&\quad \text{shuffle training data} \\
&\quad \text{for } i \text{ in np.arange}(0, n, \text{batch}_\text{size}): \\
&\quad \quad \text{sample } N=\text{np.arange}(i, i + \text{batch}_\text{size}) \\
&\quad \quad \text{calc approx derivative } DJ(\theta) = [DJ_{\theta_0}(\theta), \; DJ_{\theta_1}(\theta), \ldots, \; DJ_{\theta_n}(\theta)] \\
&\quad \quad \theta = \theta - \alpha DJ(\theta)
\end{align*}
\]

At iteration 1 of inner loop the mini-batch \( N_1 = \{1, 2, \ldots, \text{batch}_\text{size}\} \).

Suppose we re-run above code many times. Due to the shuffling, the training points in mini-batch \( N_1 \) are selected uniformly at random without replacement.

But the same training data point cannot appear in two consecutive mini-batches, so this shuffling sampling strategy is not quite the same as previous sampling without replacement strategy.

E.g. with \( m = 10 \) data points and mini-batch size 5 we might choose mini-batches:

\[
\{5, 3, 1, 2, 4\}, \quad \{8, 10, 6, 7, 9\}
\]

Hard to analyse, but in practice behaves much the same as the “sample uniformly at random with replacement and treat mini-batches independently” strategy.
Sampling Strategy

* To be avoided: select each training point once over an epoch, but in **fixed order**. E.g. its what this code does:

\[ \theta = \theta_0, \text{ batch_size} = 5, n = \#\text{training data points} \]

```python
for k in range(num_iters):
    shuffle_training_data
    for i in np.arange(0, n, batch_size):
        sample N=np.arange(i, i + batch_size)
        calc approx derivative \( DJ(\theta) = [DJ_{\theta_0}(\theta), DJ_{\theta_1}(\theta), \ldots, DJ_{\theta_n}(\theta)] \)
        \[ \theta = \theta - \alpha DJ(\theta) \]
```

* Note the lack of a shuffle at the start of each run of the inner loop.

* Suppose our training data was ordered so that all the points of one type (e.g. all the examples with label +1) come first, then all points of a second type come next (e.g. all the examples with label -1) and so on.