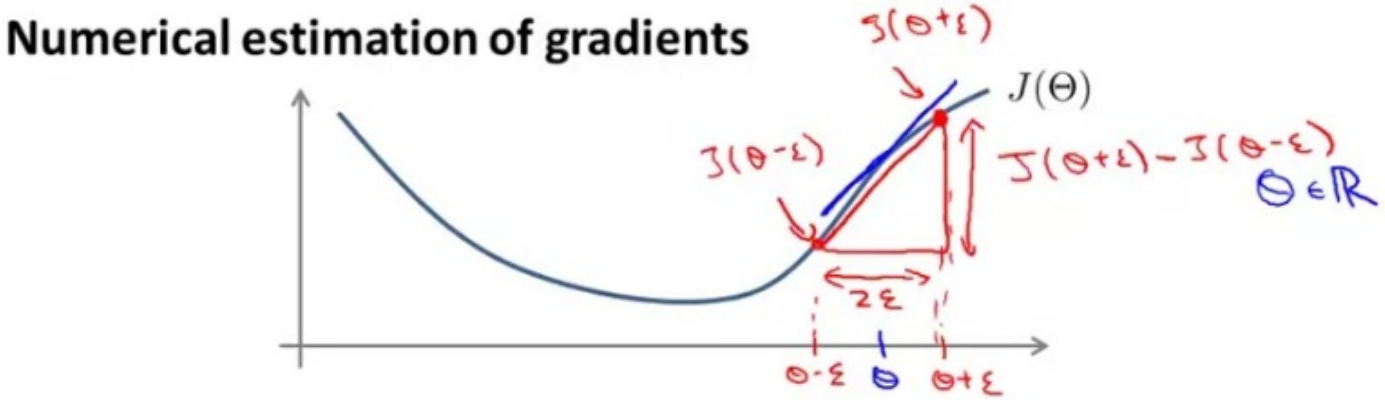
Gradient Checking

- Backpropagation has a lot of details, small bugs can be present and ruin it.
  - This may mean it looks like $J(\Theta)$ is decreasing, but in reality it may not be decreasing by as much as it should.
- So using a numeric method to check the gradient can help diagnose a bug.
  - Gradient checking helps make sure an implementation is working correctly.

Suppose that we have the function $J(\Theta)$ and we have some value theta and for this example we are gonna assume that theta is just a real number $\Theta \in \mathbb{R}$, and let's say that we want to estimate the derivative of the function at some point, so the derivative is equal to the slope of that tangent one. We're going to numerically approximate the derivative, or rather the procedure for numerically approximating the derivative.



We are going to compute $\Theta + \epsilon$ and $\Theta - \epsilon$, join them by a straight line and we're gonna use the slope of that line as an approximation to the derivative (Mathematically, the slope of the red line is this vertical height divided by this horizontal width).

So we can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

Usually, epsilon is pretty small ($\epsilon = 10^{-4}$), guarantees that the math works out properly. if epsilon becomes really small then the term becomes the slopes derivative and if the value for ϵ is too small, we can end up with numerical problems. The before equation is called **the two sided difference** (as opposed to **one sided difference**, which would be $\frac{J(\Theta+\epsilon)-J(\Theta)}{\epsilon}$).

The implementation of the gradient approximation in Octave/Matlab would be:

```
gradApprox = (J(theta + EPSILON) - J(theta - EPSILON)) / (2 * EPSILON)
```

Video Question: Let $J(\theta) = \theta^3$. Furthermore, let $\theta = 1$ and $\epsilon = 0.01$. You use the formula:

$$\frac{J(\theta+\epsilon)-J(\theta-\epsilon)}{2\epsilon}$$

to approximate the derivative. What value do you get using this approximation? (When $\theta = 1$, the true, exact derivative is $\frac{d}{d\theta}J(\theta) = 3$.

- 3.0000

> 3.0001

- 3.0301
- 6.0002

Parameter Vector θ

Now let's look at a more general case of when theta is a vector parameter $\theta \in \mathbb{R}^n$ (E.g. θ is "unrolled" version of $\Theta^{(1)}$, $\Theta^{(2)}$, $\Theta^{(3)}$), theta is a vector that has n elements:

$\theta = \theta_1, \theta_2, \theta_3, ..., \theta_n$

So if θ is a vector with n elements we can use a similar approach to look at the partial derivatives

$$\frac{\partial}{\partial\theta_1}J(\theta) \approx \frac{J(\theta_1+\epsilon,\theta_2,\theta_3,\ldots,\theta_n)-J(\theta_1-\epsilon,\theta_2,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

$$\frac{\partial}{\partial\theta_2}J(\theta) \approx \frac{J(\theta_1,\theta_2+\epsilon,\theta_3,\ldots,\theta_n)-J(\theta_1,\theta_2-\epsilon,\theta_3,\ldots,\theta_n)}{2\epsilon}$$

$$\vdots$$

$$\frac{\partial}{\partial\theta_n}J(\theta) \approx \frac{J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n+\epsilon)-J(\theta_1,\theta_2,\theta_3,\ldots,\theta_n-\epsilon)}{2\epsilon}$$

Concretely these equations give us a way to numerically approximate the partial derivative of J with respect to any one of our parameters theta. So, in octave we use the following code the numerically compute the derivatives:

```
for i = 1:n,
    thetaPlus = theta;
    thetaPlus(i) = thetaPlus(i) + EPSILON;
    thetaMinus = theta;
    thetaMinus(i) = thetaMinus(i) - EPSILON;
    gradApprox(i) = (J(thetaPlus) - J(thetaMinus)) / (2 * EPSILON);
```

- So on each loop `thetaPlus = theta` except for `thetaPlus(i)` .
  - Resets thetaPlus on each loop.
- Create a vector of partial derivative approximations (`gradApprox` $\approx$ `DVec`).
- Using the vector of gradients from backprop (DVec).
  - Check that gradApprox is basically equal or approximately equal to DVec.
  - Gives confidence that the Backpropagation implementation is correct.

Implementation Note:

- Implement backprop to compute `DVec` (unrolled $D^{(1)}$, $D^{(2)}$, $D^{(3)}$).
- Implement numerical gradient check to compute gradApprox.
- Make sure they give similar values.

- Turn off gradient checking. Using backprop code for learning.

Important:

- Be sure disable your gradient checking code before training your classifier. If you run numerical gradient computation on every iteration of gradient descent (or in the inner loop of `costFunction(...)`) your code will be very slow.

Video Question: What is the main reason that we use the backpropagation algorithm rather than the numerical gradient computation method during learning?

- The numerical gradient computation method is much harder to implement.

  > The numerical gradient algorithm is very slow.

- Backpropagation does not require setting the parameter EPSILON.
- None of the above.

Summary

Gradient checking will assure that our backpropagation works as intended. We can approximate the derivative of our cost function with:

$$\frac{\partial}{\partial\Theta}J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative **with respect to** Θ_j as follows:

$$\frac{\partial}{\partial\Theta_j}J(\Theta) \approx \frac{J(\Theta_1, \dots, \Theta_j + \epsilon, \dots, \Theta_n) - J(\Theta_1, \dots, \Theta_j - \epsilon, \dots, \Theta_n)}{2\epsilon}$$

A small value for ϵ (epsilon) such as $\epsilon = 10^{-4}$, guarantees that the math works out properly. If the value for ϵ is too small, we can end up with numerical problems.

Hence, we are only adding or subtracting epsilon to the Θ_j matrix. In octave we can do it as follows:

```
epsilon = 1e-4;
for i = 1:n,
  thetaPlus = theta;
  thetaPlus(i) += epsilon;
  thetaMinus = theta;
  thetaMinus(i) -= epsilon;
  gradApprox(i) = (J(thetaPlus) - J(thetaMinus))/(2*epsilon)
end;
```

We previously saw how to calculate the deltaVector. So once we compute our gradApprox vector, we can check that gradApprox $\approx$ deltaVector.

Once you have verified **once** that your backpropagation algorithm is correct, you don't need to compute gradApprox again. The code to compute gradApprox can be very slow.