Lecture 5

Gradient Descent and Linear Regression

Dr. David Zmiaikou

Gradient Descent

Optimisation

Often we need to solve a number of optimisation problems:

- find the best model for a certain situation
- minimise the error of its predictions
- maximise the likelihood of the data

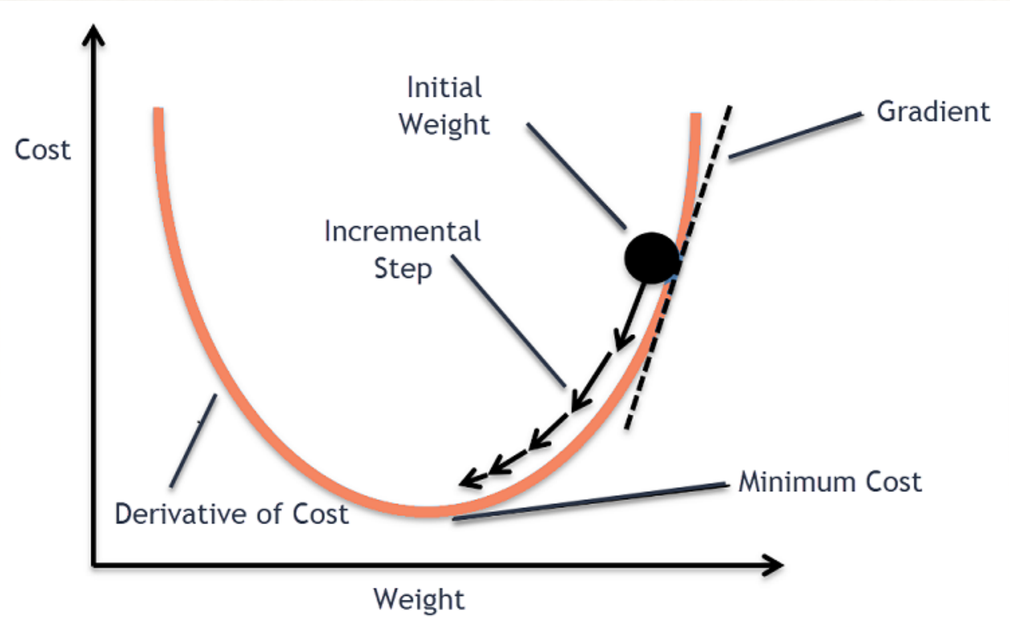
Example 1

```
def sum_of_squares(v: Vector) -> float:
    """Computes the sum of squared elements in v"""
    return dot(v, v)
```

Suppose we have some function f that takes as input a vector of real numbers and outputs a single real number.

How to maximise or minimise such a function?

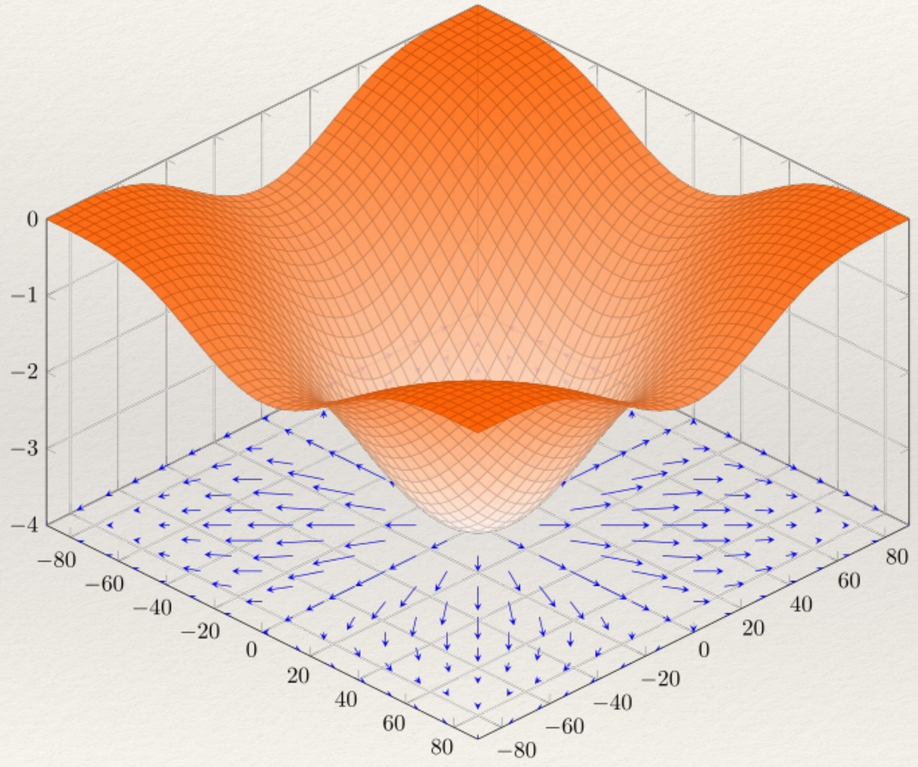
Gradient Descent



One approach to minimising a function is:

- to pick a random starting point,
- compute the gradient,
- take a small step in the direction of the gradient (i.e., the direction that causes the function to decrease the most),
- repeat with the new starting point.

Gradient of a function



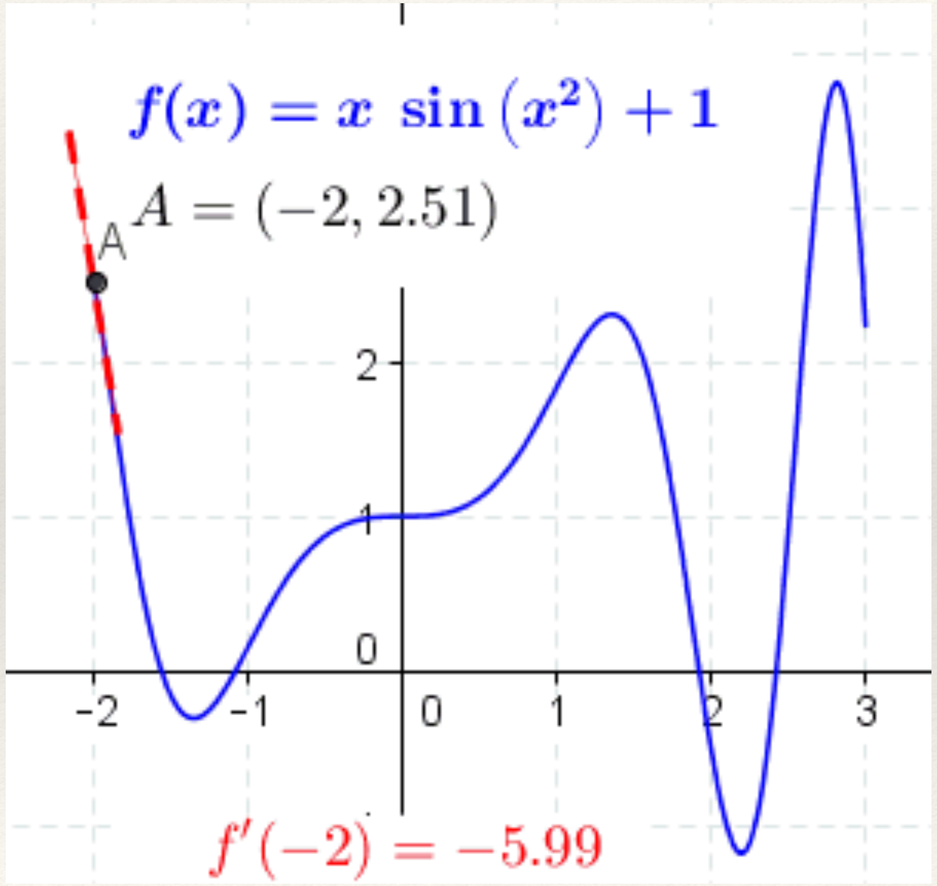
The **gradient** of a one-variable function $f : \mathbb{R} \to \mathbb{R}$ is simply its derivative:

$$\nabla f(x) = f'(x)$$

The **gradient** of an n-variable function $f : \mathbb{R}^n \to \mathbb{R}$ is the vector of its **partial derivatives**:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ \ldots, \frac{\partial f}{\partial x_n}(x) \right]$$

Derivative of a function



If f is a function of one variable, its **derivative at a point** x measures how $f(x)$ changes when we make a very small change to x.

Formally, the **derivative** is defined as the limit of the difference quotients:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Derivative of a function

```
from typing import Callable

def difference_quotient(f: Callable[[float], float],
                        x: float,
                        h: float) -> float:
  return (f(x + h) - f(x)) / h
```

Formally, the **derivative** is defined as the limit of the difference quotients:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

Example 2

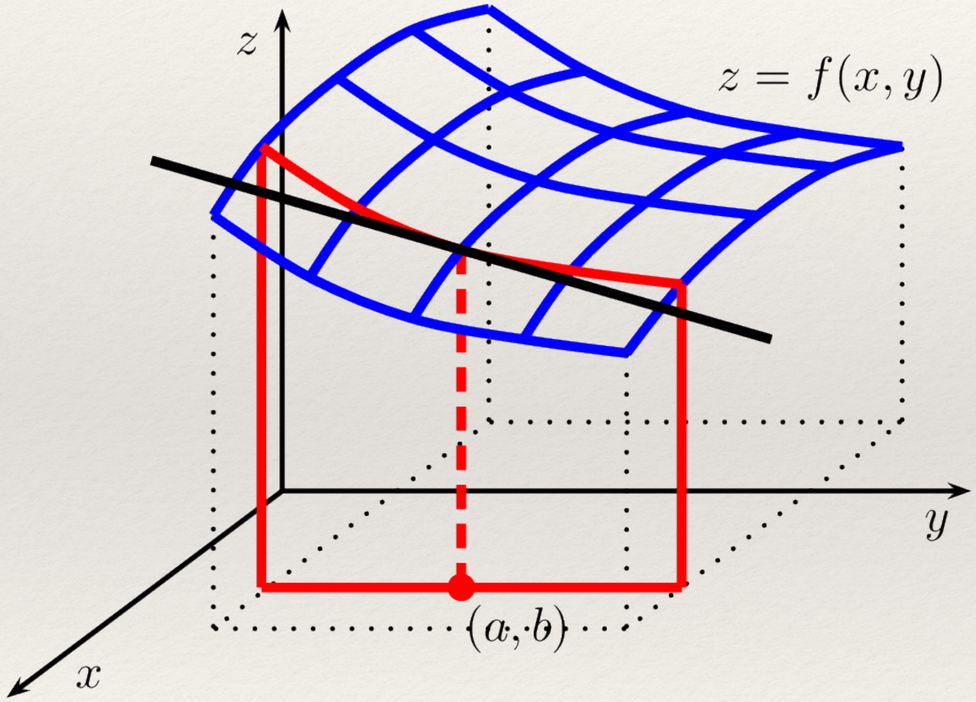
```
def square(x: float) -> float:
  return x * x

def derivative(x: float) -> float:
  return 2 * x
```

For many functions it's easy to exactly calculate derivatives. For example,

$$(x^2)' = 2x$$

Partial derivatives of a function



When $f : \mathbb{R}^n \to \mathbb{R}$ is a function of many variables, it has multiple **partial derivatives**, each indicating how f changes when we make small changes in just one of the input variables.

Formally, the ***i*-th partial derivative** is defined as the limit of the difference quotients with respect to the i-th coordinate:

$$\frac{\partial f}{\partial x_i}(x_1, \ldots, x_i, \ldots, x_n) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

Partial derivatives of a function

```
def partial_difference_quotient(f: Callable[[Vector], float],
                                v: Vector,
                                i: int,
                                h: float) -> float:
  """Returns the i-th partial difference quotient of f at v"""

  # add h to just the ith element of v
  w = [v_j + (h if j == i else 0) for j, v_j in enumerate(v)]
  return (f(w) - f(v)) / h
```

The ***i*-th partial derivative** is defined as the limit of the difference quotients with respect to the *i*-th coordinate:

$$\frac{\partial f}{\partial x_i}(x_1, \ldots, x_i, \ldots, x_n) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

Gradient of a function

```
def estimate_gradient(f: Callable[[Vector], float],
                      v: Vector,
                      h: float = 0.0001):
    return [partial_difference_quotient(f, v, i, h)
            for i in range(len(v))]
```

The **gradient** of an n-variable function $f : \mathbb{R}^n \to \mathbb{R}$ is the vector of its **partial derivatives**:

$$\nabla f(x) = \left[\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ \ldots, \frac{\partial f}{\partial x_n}(x) \right]$$

Example 3: Using the Gradient

```
import random
def gradient_step(v: Vector, gradient: Vector, step_size: float) -> Vector:
    """Moves `step_size` in the `gradient` direction from `v`"""
    assert len(v) == len(gradient)
    step = scalar_multiply(step_size, gradient)
    return add(v, step)

def sum_of_squares_gradient(v: Vector) -> Vector:
    return [2 * v_i for v_i in v]

# pick a random starting point
v = [random.uniform(-10, 10) for i in range(3)]
for epoch in range(1000):
    grad = sum_of_squares_gradient(v) # compute the gradient at v
    v = gradient_step(v, grad, -0.01)       # take a negative gradient step
    print(epoch, v)

assert distance(v, [0, 0, 0]) < 0.001       # v should be close to 0
```

It's easy to see that the **sum_of_squares** function is smallest when its input v is a vector of zeros. But imagine we didn't know that.

Use gradients to find the minimum. Pick a random starting point and take small steps in the opposite direction of the gradient until we reach a point where the gradient is very small.

Gradient Descent: Choosing the Right Step Size

The idea of moving against the gradient is clear, how far to move is not that clear. Indeed, choosing the right step size is more of an art than a science. Popular options include:

1. Using a fixed step size.

2. Gradually shrinking the step size over time.

3. At each step, choosing the step size that minimises the value of the objective function.

Gradient Descent: Choosing the Right Step Size

What if we take the step

— too small?

— too big?

Gradient Descent: Fitting Models

Often, we use gradient descent to fit parameterised models to data. In the usual case, we have

- some **dataset**,
- some (hypothesised) **model** for the data that depends (in a differentiable way) on one or more parameters,
- a **loss function** that measures how well the model fits our data (smaller is better).

We can use gradient descent to find the model parameters that make the loss as small as possible.

Example 4: Fitting a model

```
# x ranges from -50 to 49, y is always 20 * x + 5
inputs = [(x, 20 * x + 5) for x in range(-50, 50)]

def linear_gradient(x: float, y: float, theta: Vector) -> Vector:
    slope, intercept = theta
    predicted = slope * x + intercept # prediction of the model

    error = (predicted - y)         # error is (predicted - actual).
    squared_error = error ** 2   # we'll minimise squared error
    grad = [2 * error * x, 2 * error] # using its gradient.
    return grad
```

Here, we know the parameters of the linear relationship between x and y.

But imagine we would like to learn them from the data.

We'll use gradient descent to find the slope and intercept that minimise the average squared error.

Example 4: Fitting a model

```
# Start with random values for slope and intercept
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
learning_rate = 0.001

for epoch in range(5000):
    # Compute the mean of the gradients
    grad = vector_mean([linear_gradient(x, y, theta)
                        for x, y in inputs])
    # Take a step in that direction
    theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

The previous computation was for a single data point. For the whole dataset we'll look at the **mean squared error**. And the gradient of the mean squared error is just the mean of the individual gradients.

What we should do:

1. Start with a random value for theta.
2. Compute the mean of the gradients.
3. Adjust theta in that direction.
4. Repeat.

Gradient Descent: Fitting Models

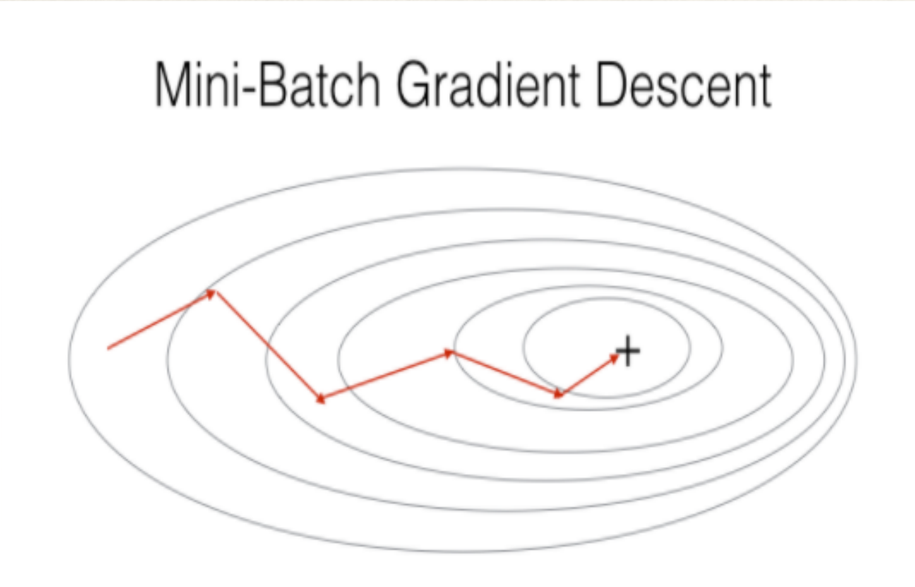
Question. What is a potential drawback of the preceding approach?

Gradient Descent: Fitting Models

Question. What is a potential drawback of the preceding approach?

Answer. Evaluation of the gradients on the entire dataset before taking a gradient step — expensive gradient computations.

Minibatch Gradient Descent



In the technique called **minibatch gradient descent** we compute the gradient (and take a gradient step) based on a "minibatch" sampled from the larger dataset.

Example 5: Fitting a model with minibatches

```
from typing import TypeVar, List, Iterator
T = TypeVar('T') # this allows us to type "generic" functions
def minibatches(dataset: List[T],
                batch_size: int,
                shuffle: bool = True) -> Iterator[List[T]]:
    # Generates `batch_size`-sized minibatches from the dataset
    # start indexes: 0, batch_size, 2 * batch_size, ...
    batch_starts = [start for start in
                    range(0, len(dataset), batch_size)]

    if shuffle: random.shuffle(batch_starts) # shuffle the batches

    for start in batch_starts:
        end = start + batch_size
        yield dataset[start:end]
```

In the technique called **minibatch gradient descent** we compute the gradient and take a gradient step based on a "minibatch" sampled from the larger dataset.

`TypeVar(T)` creates a "generic" function. Our dataset can be a list of any single type — strs, ints, lists, etc.

Example 5: Fitting a model with minibatches

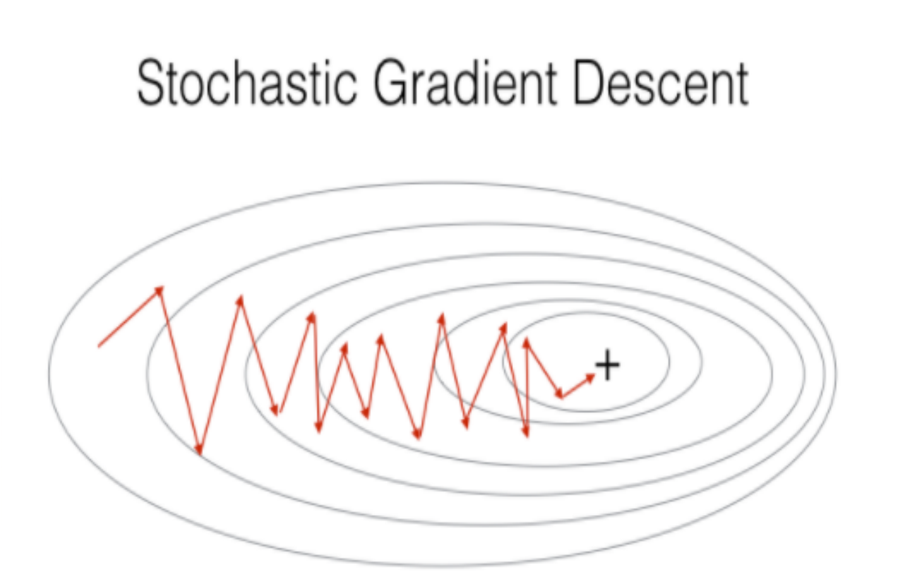
```
theta = [random.uniform(-1, 1), random.uniform(-1, 1)]

for epoch in range(1000):
    for batch in minibatches(inputs, batch_size=20):
        grad = vector_mean([linear_gradient(x, y, theta)
                            for x, y in batch])
        theta = gradient_step(theta, grad, -learning_rate)
    print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

In the technique called **minibatch gradient descent** we compute the gradient and take a gradient step based on a "minibatch" sampled from the larger dataset.

Stochastic Gradient Descent



In the technique called **stochastic gradient descent** one takes gradient steps based on one training example at a time.

Example 6: Fitting a model with minibatches

```
"""Stochastic Gradient Descent"""

theta = [random.uniform(-1, 1), random.uniform(-1, 1)]
for epoch in range(100):
    for x, y in inputs:
        grad = linear_gradient(x, y, theta)
        theta = gradient_step(theta, grad, -learning_rate)
        print(epoch, theta)

slope, intercept = theta
assert 19.9 < slope < 20.1, "slope should be about 20"
assert 4.9 < intercept < 5.1, "intercept should be about 5"
```

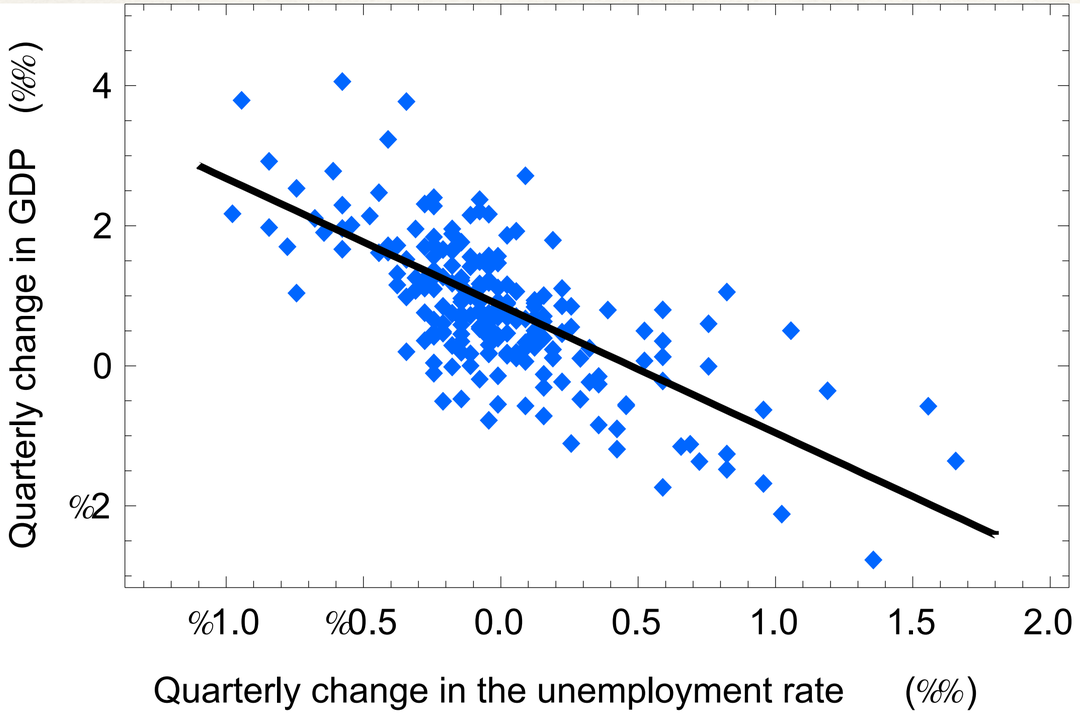
In the technique called **stochastic gradient descent** one takes gradient steps based on one training example at a time.

It descent finds the optimal parameters in a much smaller number of epochs.

Question. What are its tradeoffs?

Simple Linear Regression

Simple Linear Regression



In many applications, we need to know whether there is a relationship between two variables.

For example, we would like to check if one variable depends linearly on the other variable. **Probability theory** provides tools for that.

Now, if we know existence of a linear relation, **simple linear regression** allows us to estimate coefficients of the relation .

Example 7: DataSciencester

Let us investigate relationship between a DataSciencester user's number of friends and the amount of time the user spends on the site each day.

Suppose that we have noticed that having more friends causes people to spend more time on the site.

Example 7: DataSciencester

We hypothesise that there are constants α (alpha) and β (beta) such that

$$y_i = \beta x_i + \alpha + \varepsilon_i$$

where y_i is the number of minutes user i spends on the site daily,
x_i is the number of friends of the user i,
ε_i is a hopefully small error term representing the fact that there are other factors not accounted for by this simple model.

Example 7: DataSciencester

```
def predict(alpha: float, beta: float, x_i: float) -> float:
    return beta * x_i + alpha
```

If we know the constants α and β, then we can make predictions for the number of minutes y_i .

Example 7: DataSciencester

```
def error(alpha: float, beta: float, x_i: float, y_i: float) -> float:
    """
    The error from predicting beta * x_i + alpha
    when the actual value is y_i
    """
    return predict(alpha, beta, x_i) - y_i
```

But if we don't know them, how do we choose α and β?

Well, any choice of alpha and beta gives us a predicted output for each input x_i. Since we know the actual output y_i, we can compute the error for each pair.

Example 7: DataSciencester

```
def sum_of_sqerrors(alpha: float, beta: float,
                    x: Vector, y: Vector) -> float:
    return sum(error(alpha, beta, x_i, y_i) ** 2
               for x_i, y_i in zip(x, y))
```

We would like to know is the total error over the entire dataset. But we don't want to just add the errors — if the prediction for x_1 is too high and the prediction for x_2 is too low, the errors may just cancel out.

Instead we add up the **squared errors**.

Example 7: DataSciencester

```
def least_squares_fit(x: Vector, y: Vector) -> Tuple[float, float]:
    """
    Given two vectors x and y,
    find the least-squares values of alpha and beta
    """
    beta = np.corrcoef(x, y) * np.std(y) / np.std(x)
    alpha = np.mean(y) - beta * np.mean(x)
    return alpha, beta
```

The **least squares solution** is to choose α and β that make sum_of_sqerrors as small as possible.

Thank you!