[**Coding Deep Learning for Beginners — Linear Regression (Part 3): Training with Gradient Descent**](#_e6di0ti5pa8r) **2**

[**Goal**](#_8x3ti5e19hp1) **3**

[**Training by Brute Force**](#_1z58k95q4gks) **3**

[Conclusions](#_g29fc2le62ot) 13

[**Role of the Derivative**](#_l5hu7xs4hu5n) **14**

[Intuition](#_r4ha1v1jp1tl) 15

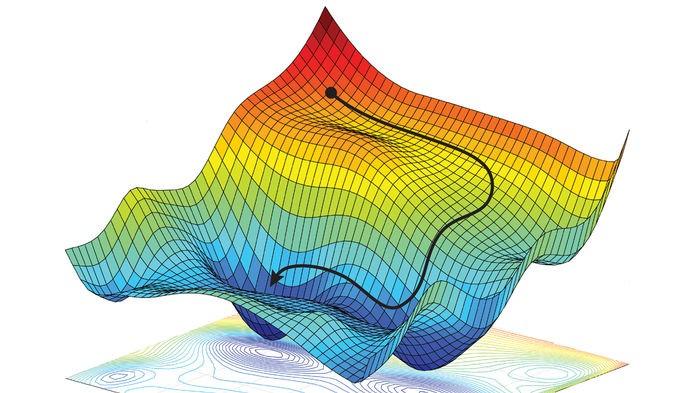
[**Gradient Descent**](#_qjechisvx36w) **19**

[Geometrical Iterpretation](#_yvrn9m27ohrh) 24

[Update Rule](#_hre7jxr2kjhn) 31

[Implementation](#_5le1zrmp18ug) 32

[**Summary**](#_9xw301nieazw) **38**



Source: <https://reconsider.news/2018/05/09/ai-researchers-allege-machine-learning-alchemy/>

# Coding Deep Learning for Beginners — Linear Regression (Part 3): Training with Gradient Descent



[Kamil Krzyk](https://towardsdatascience.com/@krzyk.kamil?source=post_page-----fcd5e0fc077d----------------------)

Follow

[Aug 23, 2018](https://towardsdatascience.com/coding-deep-learning-for-beginners-linear-regression-gradient-descent-fcd5e0fc077d?source=post_page-----fcd5e0fc077d----------------------) · 12 min read

This is the 5th article of series “**Coding Deep Learning for Beginners**”. You will be able to find here *links to all articles*, *agenda, and general information about an estimated release date of next articles* [*on the bottom of the 1st article*](https://medium.com/@krzyk.kamil/coding-deep-learning-for-beginners-start-a84da8cb5044)*.* They are also available in my [open source portfolio — **MyRoadToAI**](https://github.com/FisherKK/F1sherKK-MyRoadToAI), along with some mini-projects, presentations, tutorials and links.

# **Goal**

In this article, I will **explain the concept of training Machine Learning algorithms with Gradient Descent**. Majority of supervised algorithms are taking advantage of it — especially all Neural Networks. It is a **crucial topic** and also one of the most substantial obstacles to overcome for people starting with Machine Learning. This is because it’s **based on Calculus** and some people didn’t cover it at school, or their knowledge got rusty.

But don’t worry even **if you aren’t comfortable with math or don’t know Calculus at all you can still comprehend it**, learn it and use it. I will show you how!

# **Training by Brute Force**

Thanks to the Cost Function that was introduced in the previous article it is already possible to train the model to predict the prices of Cracow apartments. The **model is using a size of the apartment only**, so its form isn’t too complicated:



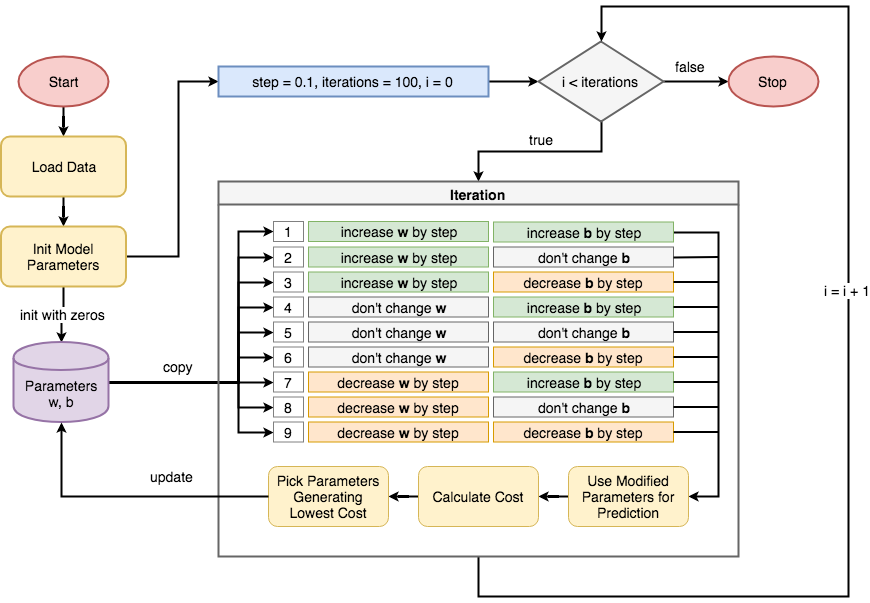
where:

* *ŷ* — predicted apartment price,
* *x*—apartment size,
* *w*— weight assigned to apartment size,
* *b*— bias.

Subscript “0” stands for position of feature in each vector of dataset *X* and is omitted to improve readability.

The goal of the training process will be **finding the combination of *w* and *b* parameters, for which the Cost Function returns as small error as possible**.

To improve model performance, very naive thing will be done. Parameters will be changed by a small fixed **step**, in many **iterations**, and as it’s unknown if parameter values should be changed at all and in which **direction**. Consequently, cost value for all possible combinations will be calculated to find the best modification and that’s why it’s called “training by brute force.” **Three actions** can be performed on the parameter during each training iteration: **increase, decrease, do not change at all.** There are **two parameters**, which means every iteration there will be **3²= 9 sets of parameters to check**, for which predictions will be made, and the cost will be calculated. The model will be updated with the settings for which cost has the lowest value.



Training with Brute Force — diagram showing how to train model with two parameters (w and b) without using Gradient Descent algorithm.

Let’s implement that and see the result. Majority of operations were already coded in last articles, therefore only training loop is left to be implemented.

Full code available under this [link](https://gist.github.com/FisherKK/7c19abce0bab8c2a90384f632cc71c52#file-a5_brute_force_training_full-py).

There are only two parameters and code is already complicated enough to omit some parts to improve readability. Hidden parts can be implemented very similarly as shown one — the only difference is that mathematical operator changes from += to -= or sometimes is not used at all.

Snippet starts with an import of deepcopy function out of copy module. It is needed to duplicate model parameters stored in the model\_parameters dictionary, along with its content. Before any training happens, model\_parameters are used by function predict and mse to measure the potential of the model by calculating it’s initial error.

Then model\_parameters are copied nine times. Each copy has modified by step value in a different way (addition, subtraction, no change) and then used for predictions. Depending on candidate\_pred correctness, the copy has cost value assigned to it as candidate\_error variable. Error and parameters are stored in candidates and errors lists. After every candidate is calculated, and their performance is measured, then set of parameters for which cost was the lowest is used to replace model\_parameters . It happens iterations times which is set to 100 by default. Notice that function train is not returning any result and model\_parameters are modified in-place.

After all of this, the train function can be used on loaded data and initialized model parameters in order to find the values that will minimize the error:

It is possible to add simple print functions into train function body and display how training process is changing model parameters (check this [link](https://gist.github.com/FisherKK/7c19abce0bab8c2a90384f632cc71c52#file-a5_brute_force_training_full-py) for full code if you didn’t do it before):

Initial state:

- error: [75870.4884482]

- parameters: {'w': array([0.]), 'b': 0.0}

Iteration 0:

- error: [73814.2609893]

- parameters: {'w': array([0.1]), 'b': 0.1}

Iteration 20:

- error: [38764.28631114]

- parameters: {'w': array([2.1]), 'b': 2.1}

Iteration 40:

- error: [15284.92972772]

- parameters: {'w': array([4.1]), 'b': 4.1}

Iteration 60:

- error: [3376.19123904]

- parameters: {'w': array([6.1]), 'b': 6.1}

Iteration 80:

- error: [1753.32046443]

- parameters: {'w': array([7.1]), 'b': 8.1}

Final state:

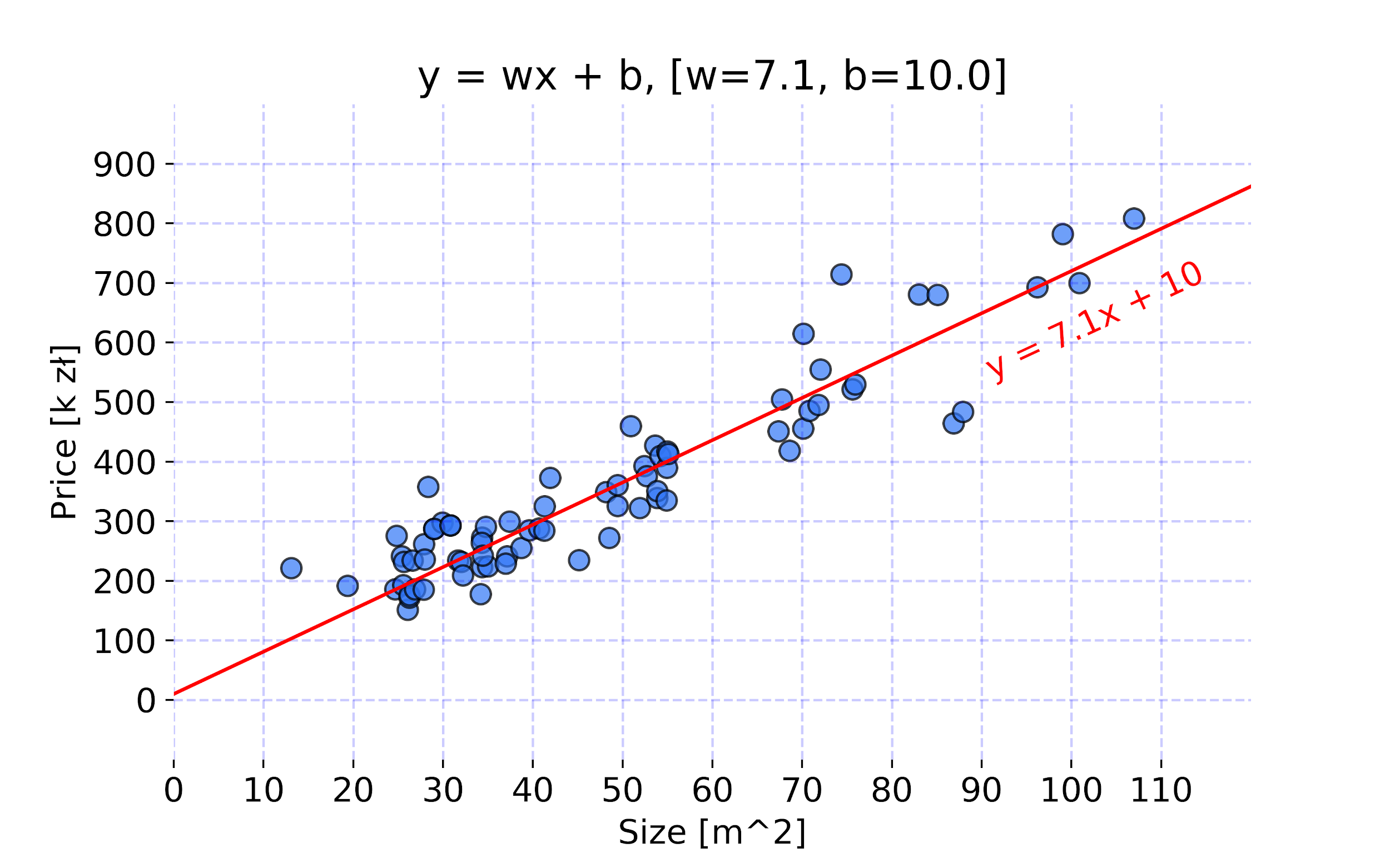
- error: [1741.85716443]

- parameters: {'w': array([7.1]), 'b': 10.0}

**Error is decreasing with each iteration.** This is because all training samples are used during each iteration — but more about this in the future articles. Another important observation is that **error value does not equal zero after training was finished.** It is impossible to go through all the points with just linear function.

Finally it is possible to display the results of training and see how model is capable of approximating apartment prices:





Model projection after training. Code used to create the chart is available [here](https://gist.github.com/FisherKK/55e4a1d52df1951fdb2ba4a7ba89ffa2).

## **Conclusions**

No fancy technique was needed to train such simple model and find parameters that can perform well on given data. But **current method comes with many limitations**:

* A lot of **boilerplate code**.
* **Too many combinations have to be tested during each iteration.** This model is using only two parameters. Imagine what would happen in models like Neural Networks where amount of parameters is sometimes counted in millions — 3¹ ⁰⁰⁰ ⁰⁰⁰ isn’t a nice number…
* **Step** by which parameters are change is **fixed for every parameter**. **Range of its value is also unknown,** sometimes 0.1 might be too much, sometimes it might be to small.
* **Direction in which each parameter should be changed is unknown,** that’s why all possible changes has to be tried out. It takes a lot of computational power and is very slow.

# **Role of the Derivative**

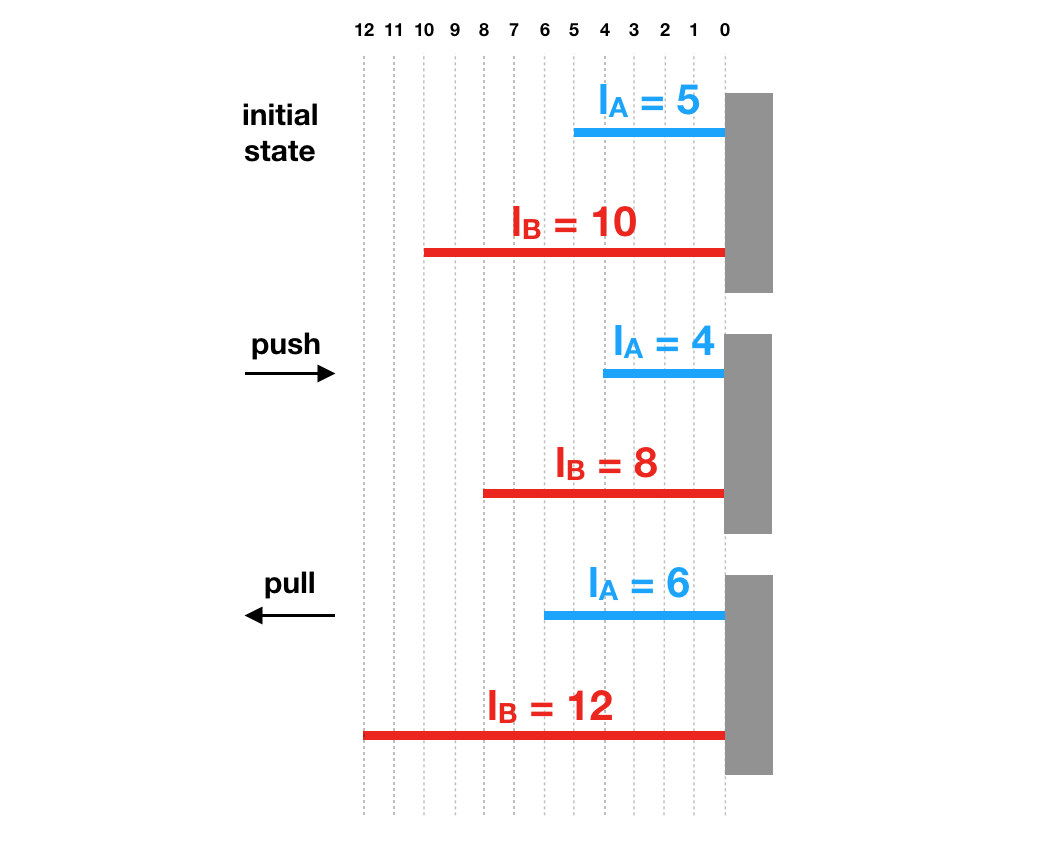
Paradoxically, understanding **what Derivatives can be used for is more important rather than being able to calculate it** when it comes to reusing already invented Machine Learning concepts that are involving Gradient Descent. [Andrew Trask](https://twitter.com/iamtrask?ref_src=twsrc%5Egoogle%7Ctwcamp%5Eserp%7Ctwgr%5Eauthor) in his book [“Grokking Deep Learning”](https://www.manning.com/books/grokking-deep-learning) have provided very nice intuition of what derivative is and how it works. Explanation of this article will use similar reasoning.

## **Intuition**

Imagine two rods sticking out of the wall. Let’s name them rod A and rod B. Rod A is two times shorter than rod B. Two experiments are made in order to understand the relationship between the rods:

* Firstly, rod A is being pushed into the wall. What can be observed is that length of rob B is shrinking along with rod A but by the double amount.
* Secondly, rod A is pulled out of the wall and it length increases. As a result rod B is also being extended but two times faster.





Now let’s try to **describe the relationship between two rods in form of mathematical equation.** In both cases it is possible to observe that length of rod B is always double the amount of rod A:





where:

* *l\_A* — length of rod A,
* *l\_B* — length of rod B.

Hold on for a second and think **about the role of number 2 in that equation.** It’s a derivative of *l\_B* with respect to *l\_A* and can be written as:





What can be learned from that?

* **Derivative describes how one variable changes when the other one is modified.** So in this case, by what factor does length of rod B extends or shrinks when rod A length is modified.
* **Derivative is calculated between two variables.**
* **When derivative has positive value then both values change in the same direction.** If rod A extends then rod B extends too because 2 is a positive number. And vice versa, **if derivative has negative value then values change in opposite direction.**

Remember that:

*For differentiable function, it is possible to find relationship between the two variables — by what amount one variable changes and in what direction, when other one is modified via using derivative.*

Considering previous issues with “brute force learning”, derivatives has some interesting properties that could be used to solve the problems.

# **Gradient Descent**

An iterative optimization **algorithm for finding minimum of the** [**convex function**](https://en.wikipedia.org/wiki/Convex_function)**.** It is based on Calculus — **relies on properties of the first-derivative** to find in what **direction** and with what **magnitude coefficients of the function should be modified.**

In previous article it was mentioned that **Cost Function** used to measure Machine Learning model performance **needs to be differentiable**. If that is not the case, then Gradient Descent algorithm can’t be applied.

Let’s take Mean Squared Error function, which is widely used as a Cost Function for regression models.





* i — index of sample,
* ŷ — predicted value,
* y — expected value,
* m — number of samples in dataset.

The predicted value *ŷ* can be replaced with the formula of apartment price approximation model. Let’s also tweak used nomenclature a little bit as **it is very popular to name the error of Machine Learning model with the capital letter *J****.*

**

**

Now it is easy to see that **error value depends on *w* and *b* coefficients**. Parameters ***m*, *x*, *y* can be treated as constants which values are known and determined by the dataset** on which model is trained.

It was already said that it is possible to calculate derivative between any two function parameters and it will provide information by what factor one changes when other is modified. Consequently derivative of *J:*

* with respect to parameter *w* provides information how to tweak value of parameter *w* to either minimize or maximize value of *J,*

**

**

* with respect to parameter *b* provides information how to tweak value of parameter *b* to either minimize or maximize value of *J.*

**

**

**If the model had more parameters then more derivatives would need to be calculated.** And this is the moment where Calculus comes in handy. In great majority of cases, math is not needed for using Gradient Descent algorithm. **Derivatives of various functions used in Machine Learning, detailed calculations or code implementations are** [**globally present in the web**](https://en.wikipedia.org/wiki/Activation_function)**.**

For start it is fine to look up the answers in the web. Ability to use Calculus will for sure lead to better understanding of how various Machine Learning mechanisms works. It will also provide ability to check code of other people.

If you are curious how these two derivatives were calculated then try [this blog post](http://mccormickml.com/2014/03/04/gradient-descent-derivation/) or take a look at [106th page of Ian Goodfellow’s book “Deep Learning”](https://www.deeplearningbook.org/contents/ml.html).

## **Geometrical Iterpretation**

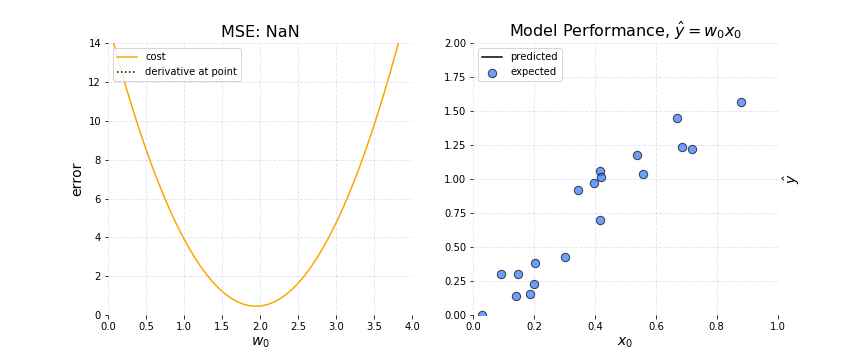
It is possible to think about the derivative as of the **slope of the tangent line to the graph at the given point**. Synonym of the word “gradient” is word “slope.”

The model parameter values are usually randomized at the start. Their values **impose location of the point** on the **error curve** (for a model with one parameter), **error surface** (for a model with two parameters) or **error hyperplane** (for more than two parameters). The goal of Gradient Descent algorithm is to **find parameter values, so the point appears always in the lowest area** because that’s where error value is the lowest.

Here are the visualizations presenting how the model with one parameter is iteratively improved on randomly generated data. There are two scenarios:

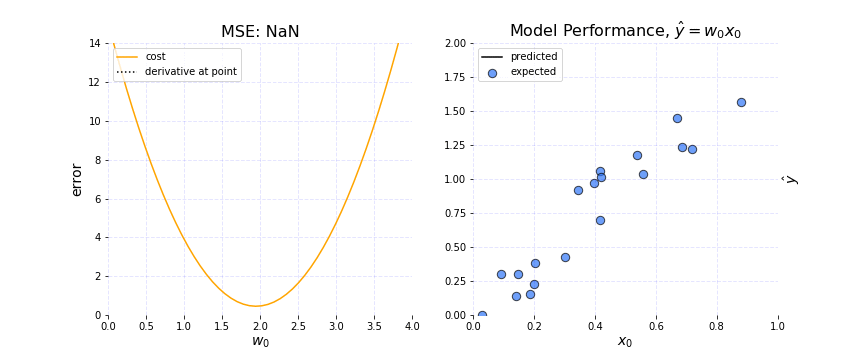
* randomized weight value is too small and needs to be increased,





* randomized weight value is too large and needs to be decreased.



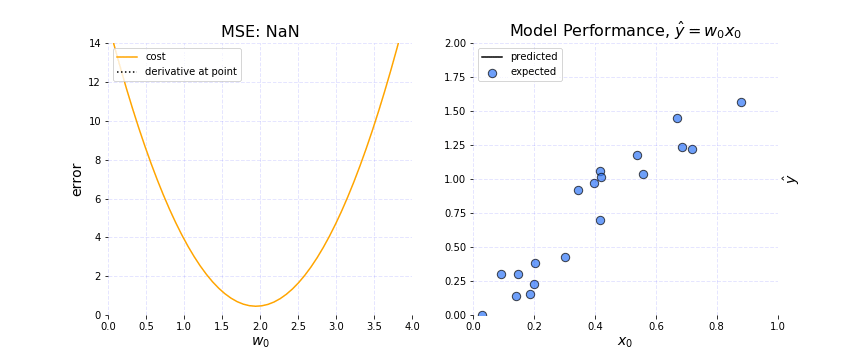


There are few critical observations:

* Notice that the **weight value is too small then the derivative is negative**, and the **weight value is too large then the derivative is positive**. Consequently, **to minimize the error function, derivative needs to be subtracted from the weight to bring the point closer to the global minimum**. If derivative were added, then the error would increase instead.
* **Derivative value is decreasing** as weight updates bring error values closer to the global minimum. It is also correct from geometrical perspective because the **“slope” of the tangent line is also becoming less steeper** with each iteration.
* The distance between points is called a **gradient step**. Notice that derivative values are much larger than weight value on axis x. If the first statement were true, subtracting derivative from weight wouldn’t change weight value by such small numbers. This is because **when updating parameters with Gradient Descent only a tiny part of the derivative needs to be used to keep numerical stability and to not jump over the global minimum**. What part of derivative should be used or how big gradient step should be is determined by **learning rate** hyperparameter.

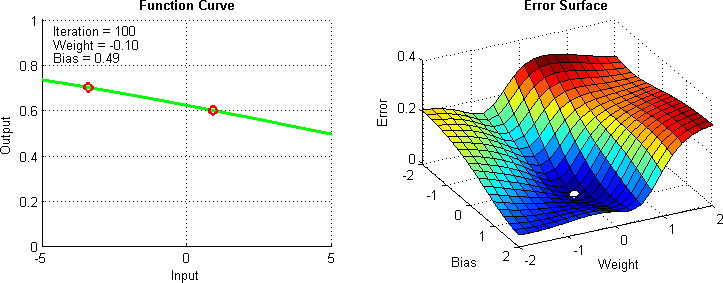
When learning rate is too range then values start to bounce from the curve and error keeps increasing infinitely:





If model is dependant from two parameters rather than one, then in order to display error value additional axis is needed (for bonus parameter). This automatically bring the visualization from 2D plane to 3D surface. Here is the good example found on the web how such error surface could look like:





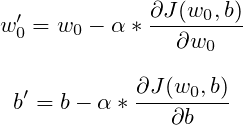
Source: <https://towardsdatascience.com/improving-vanilla-gradient-descent-f9d91031ab1d>

For more than two parameters visualization is very hard and tricky. Usually it is important to rely on dimensionality reduction methods such as [PCA](https://en.wikipedia.org/wiki/Principal_component_analysis) and related ones. Luckily math is very similar and scales to any number of parameters.

## **Update Rule**

To update model parameters, so the **convergence** is achieved, the following math has to be iteratively applied:





where:

* w’ — new weight value,
* b’ — new bias value,
* w — current weight value,
* b — current bias value,
* α — learning rate,
* dJ(w\_0, b)/dw\_0 — derivative of J with respect to w\_0,
* dJ(w\_0, b)/db— derivative of J with respect to b.

Parameters should be updated until cost function value won’t decrease anymore and the whole process can be stopped if the current model state is already satisfying.

## **Implementation**

Let’s write new train function which uses **Gradient Descent** algorithm:

Parameters are being changed by very small values as only 0.0005 of derivative value at specific point is used. It takes over 20000 iterations to achieve good results. Whole process could be speed up if input data was previously [normalized](https://en.wikipedia.org/wiki/Normalization_(statistics)).

During every iteration, partial gradient is calculated for every data sample and summed up for whole dataset. After that accumulated gradient is averaged and used to update parameters at the end of **gradient step** (iteration in which parameters are updated).

Now the new train function can be **used in exactly same way as before:**

which produces following result:

Initial state:

- error: [75870.4884482]

- parameters: {'w': array([0.]), 'b': 0.0}

Iteration 0:

- error: [13536.3070032]

- parameters: {'w': array([10.17501967]), 'b': array([0.17843399])}

Iteration 4000:

- error: [1737.28457739]

- parameters: {'w': array([7.09101188]), 'b': array([10.96966037])}

Iteration 8000:

- error: [1707.33242182]

- parameters: {'w': array([6.9583785]), 'b': array([18.67110985])}

Iteration 12000:

- error: [1692.21685452]

- parameters: {'w': array([6.86415678]), 'b': array([24.14215949])}

Iteration 16000:

- error: [1684.5886765]

- parameters: {'w': array([6.79722241]), 'b': array([28.02875048])}

Final state:

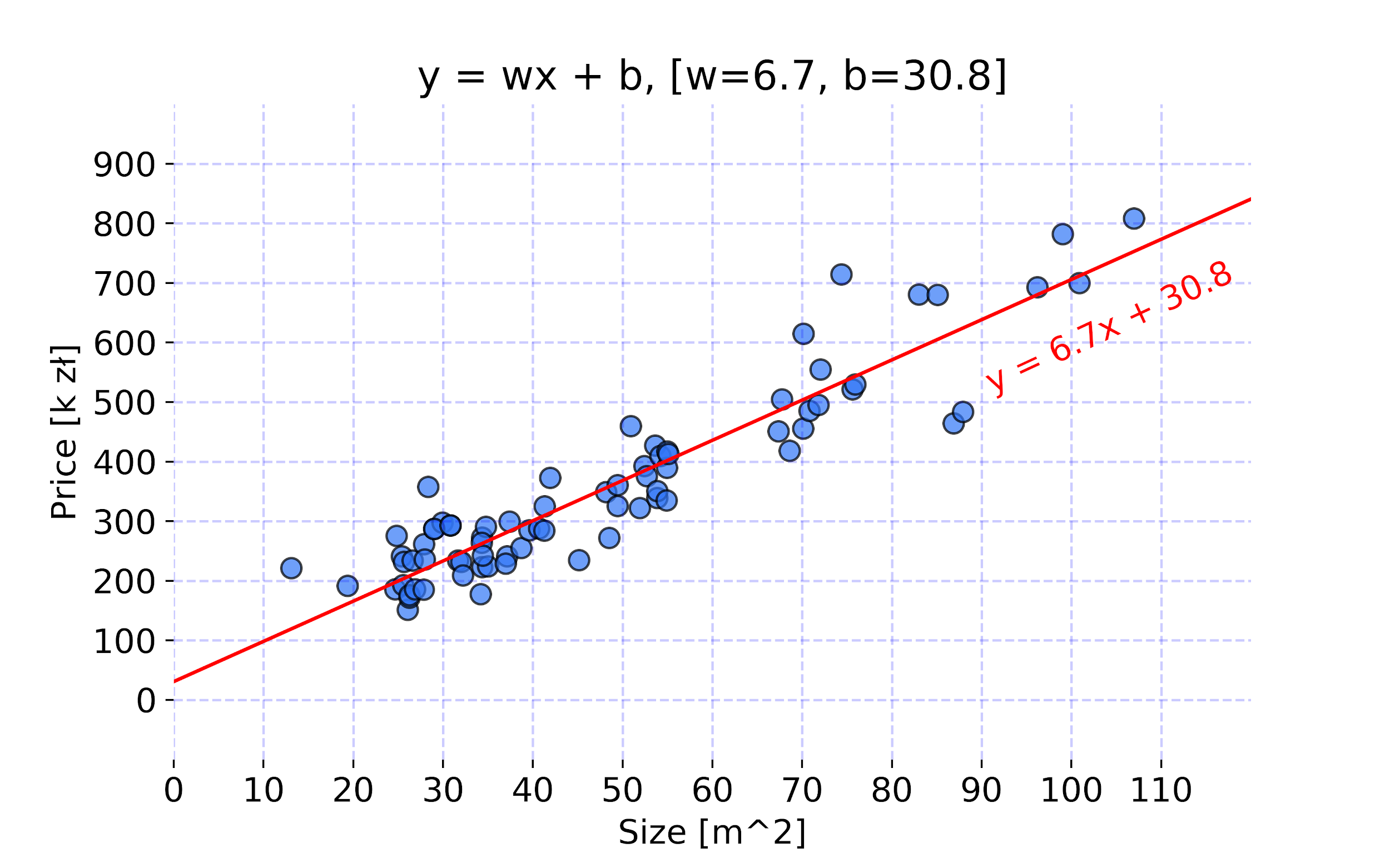
- error: [1680.73973307]

- parameters: {'w': array([6.74968272]), 'b': array([30.78917543])}

There is no point in comparing both training methods. The main problem of brute force training is it’s inefficiency. **In case of Gradient Descent algorithm, model is iteratively reaching towards convergence** **and there is no need to check all possible modifications that can be made.** It will make huge difference when dealing with complicated problems — data with high dimensionality.

Finally let’s see the model projection:





Model projection after training. Code used to create the chart is available [here](https://gist.github.com/FisherKK/ff1489616c25f2295c594040dd9212c3).

# **Summary**

We are very close to creating our first Machine Learning model — Linear Regression, written from scratch in Python, that is capable of predicting prices of Cracow’s apartments. With knowledge which have been already provided, you could create final solution on your own!

In this article I have shown to you **how training loop works** and **how cost function can be used** to find best parameters of the model. Then I’ve pointed out a lot of disadvantages that presented method had. In the next part we went through explaining **what derivative is and what it is used for**. I have told you what is **Gradient Descent algorithm from theoretical, numerical and geometrical perspectives,** usingMean Squared Error function example as a help in explanation. Finally I have **turned explained concept into code and used it for training** our model on single feature — apartment size.

What we are lacking now is… **generalization**. All concepts, code snippets are presented for simple problem where data is one dimensional. This simplification is of course intentional. In the next article, **I will show you how to refactor all those concepts so they work for any number of features and never have to be edited again!** It can be achieved thanks to matrix multiplications and whole process is called **vectorization**.