哈尔滨工业大学

<<深度学习技术>>

报告题目：

<< Introduction to TensorFlow >>

(2017 年度春季学期)

姓名：	**Kulnevich Vera**	**Petrov Aleksei**
学号：	**16SF03214**	**16SF03215**
学院：	计算机学院	计算机学院
教师：		

INTRODUCTION TO TENSORFLOW

The principles of working with tensorflow are quite simple. We must compile an operations graph, then transfer the data to this graph and give the command to perform the calculations. In the picture below you can see 3 examples of such graphs:

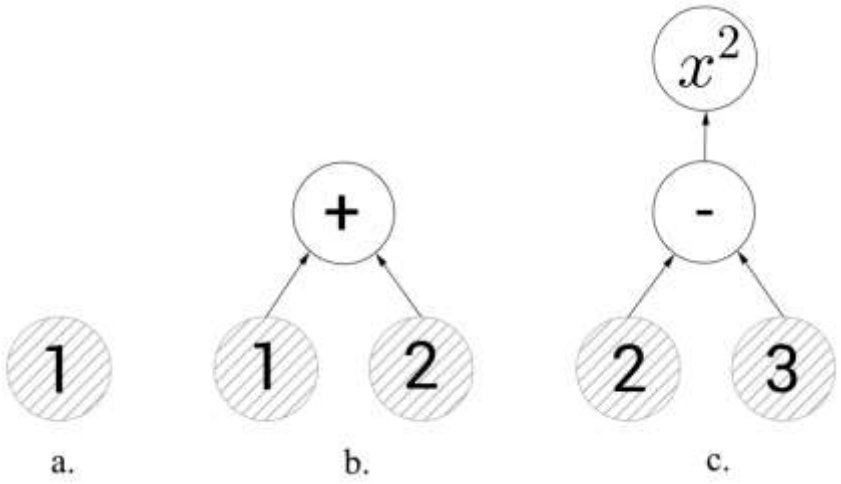

Fig 1. a. graph with one vertex, b. addition operation, c. two vertices with operations

The left graph a contains only one vertex representing a constant with a value of 1. Here and below, in such illustrations, circles with gray hatching will denote vertices with constants, and without hitting the vertex with operations. The central graph b illustrates the addition operation. If we ask tensorflow to calculate the value of the vertex representing the addition operation, then it will compute the values of the directed edges into it and sum them (that is, 3 will be returned). On the right graph c we have two vertices with operations - subtraction and squaring. If we try to calculate the vertex representing squaring, then tensorflow will first perform the subtraction.

An empty graph can be created by the function `tf.Graph ()`, in addition, the graph is created by default when the library is connected and if you do not explicitly indicate the graph, then it will be used. The example below shows how you can create two constants in two different graphs.

```
import tensorflow as tf # In the future this line will be omitted

# Save the graph by default to a variable
default_graph = tf.get_default_graph()
# Declare a constant in the default graph
c1 = tf.constant(1.0)

# Create an empty graph
second_graph = tf.Graph()
with second_graph.as_default():
    # In this block we work in the second graph
    c2 = tf.constant(101.0)

print(c2.graph is second_graph, c1.graph is second_graph) # True, Fals
e
print(c2.graph is default_graph, c1.graph is default_graph) # False, T
rue
```

Data transmission and execution of operations occur in sessions. The session is started by calling `tf.Session`, and closing it by calling the close method on the session object. You can use the `with` clause, which automatically closes the session:

```
default_graph = tf.get_default_graph()

c1 = tf.constant(1.0)

second_graph = tf.Graph()
with second_graph.as_default():
    c2 = tf.constant(101.0)

session = tf.Session() # Open the session on the default graph
print(c1.eval(session=session))
# print(c2.eval(session=session)) # So it is impossible, not that grap
h
session.close()

# the same:
with tf.Session() as session:
    print(c1.eval()) # Do not need to pass the session to a method eva
l

# Use another graph:
with tf.Session(graph=second_graph) as session:
    print(c2.eval()) # Do not need to pass the session to a method eva
l

#Conclusion:
# 1.0
# 1.0
# 101.0
```

Next, we turn to the construction of graphs. In the previous examples, constants were added to the graph and it was time to find out what it was and how they differed from placeholder and variables. In the example below, a more complex graph representing the expression $\boldsymbol{a} \cdot \boldsymbol{x} + \boldsymbol{b}$ is constructed.

```
# Declare the constant a. This constant and its value will be sewn in
the graph itself
# The declaration below lists all possible parameters, although it was
 sufficient to specify only the value:
# a = tf.constant(2.0)
# Description of parameters:
# value (The first argument) is the value of the constant
# shape - dimension. For example: [] - number, [5] - array of 5 elemen
ts, [2, 3] - matrix 2x3(2 rows of 3 columns)
# dtype - Used data type
# name - Name of the node. Allows to give a name to the node and furth
er to find the node on it
a = tf.constant(2.0, shape=[], dtype=tf.float32, name="a")
# Declare the variable x
# When declaring a variable, you can specify quite a lot of arguments
# For a full list, you can look at the documentation, I will only say
about the main:
# initial_value - Value of the variable after initialization
```

```
# dtype - тип, name - Name, as well as for constants
x = tf.Variable(initial_value=3.0, dtype=tf.float32)
# Since we usually need to transfer data to the model during the cours
e of work, the constants do not really suit us
# For input data there is a special type of placeholder
# In contrast to a constant, it does not require you to specify a valu
e in advance, but requires you to specify the type
# Also you can specify the dimension and name
b = tf.placeholder(tf.float32, shape=[])
# And declare the operation of multiplication, if desired, you can als
o specify the name
f = tf.add(tf.multiply(a, x), b) # можно было написать просто f = a*x
+ b

with tf.Session() as session:
    # First of all, you need to initialize all global variables
    # In our case this is only x
    tf.global_variables_initializer().run()
    # Please calculate the value of node f inside the session
    # In the parameter feed_dict we pass the values of all placeholder
s
    # in this case b = -5
    # The function will return a list of the values of all nodes submi
tted for execution
    result_f, result_a, result_x, result_b = session.run([f, a, x, b],
 feed_dict={b: -5})
    print("f = %.1f * %.1f + %.1f = %.1f" % (result_a, result_x, resul
t_b, result_f))
    print("a = %.1f" % a.eval()) # While the session is open, you can
compute the nodes
    # The eval method is similar to the session run method, but it doe
s not allow you to pass the input data (the feed_dict parameter)

    # Variables can be modified at runtime without using the graph:
    x = x.assign_add(1.0)
    print("x = %.1f" % x.eval())

# Conclusion:
# f = 2.0 * 3.0 + -5.0 = 1.0
# a = 2.0
# x = 4.0
```

So, the placeholder is a node through which new data will be transferred to the model, and a variable is a node that can change as the graph progresses. Now we can start learning the first model. In the previous code fragment, we have compiled the graph of the linear function $\boldsymbol{a} \cdot \boldsymbol{x} + \boldsymbol{b}$, now let's go a little further and approximate the function $\boldsymbol{a} \cdot \boldsymbol{x} + \boldsymbol{b}$ by a set of points.

THE FIRST LEARNING ALGORITHM

For tensorflow to teach the model, we need to add two more things: the loss function and the optimization algorithm itself.

The loss function is a function that takes the value of the function predicted by the model and the actual value, and returns the distance between them (we will call this value an error). For example, if we predict a real value, then as the loss function, we can take the square of the difference of the arguments or the modulus of their difference. If we have a classification problem,

then the loss function can return 0 with the correct answer and 1 with errors. Roughly speaking, the loss function must return a nonnegative real number and it must be the greater, the stronger the model is mistaken, and then the problem of learning the model will be reduced to minimization. And although the last sentence is not entirely correct, it fully reflects the idea of machine learning.

From the optimization methods, we will consider only the classical gradient descent. About him written already very much, so I will not disassemble it "by brick" and go into details (the material and so it's not small). However, you need to understand it, so I'll try to explain the method briefly and visually with the help of visualizations. Below are 2 variants of the same graph - $\boldsymbol{\sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2\right) + \cos(2x + 1)}$. The task of the method is to find a local minimum, i.e. From the point (taken at random, on the graph $\left(\frac{1}{2}; \frac{1}{2}\right)$) to get into the depression (the blue zone on the graphs).

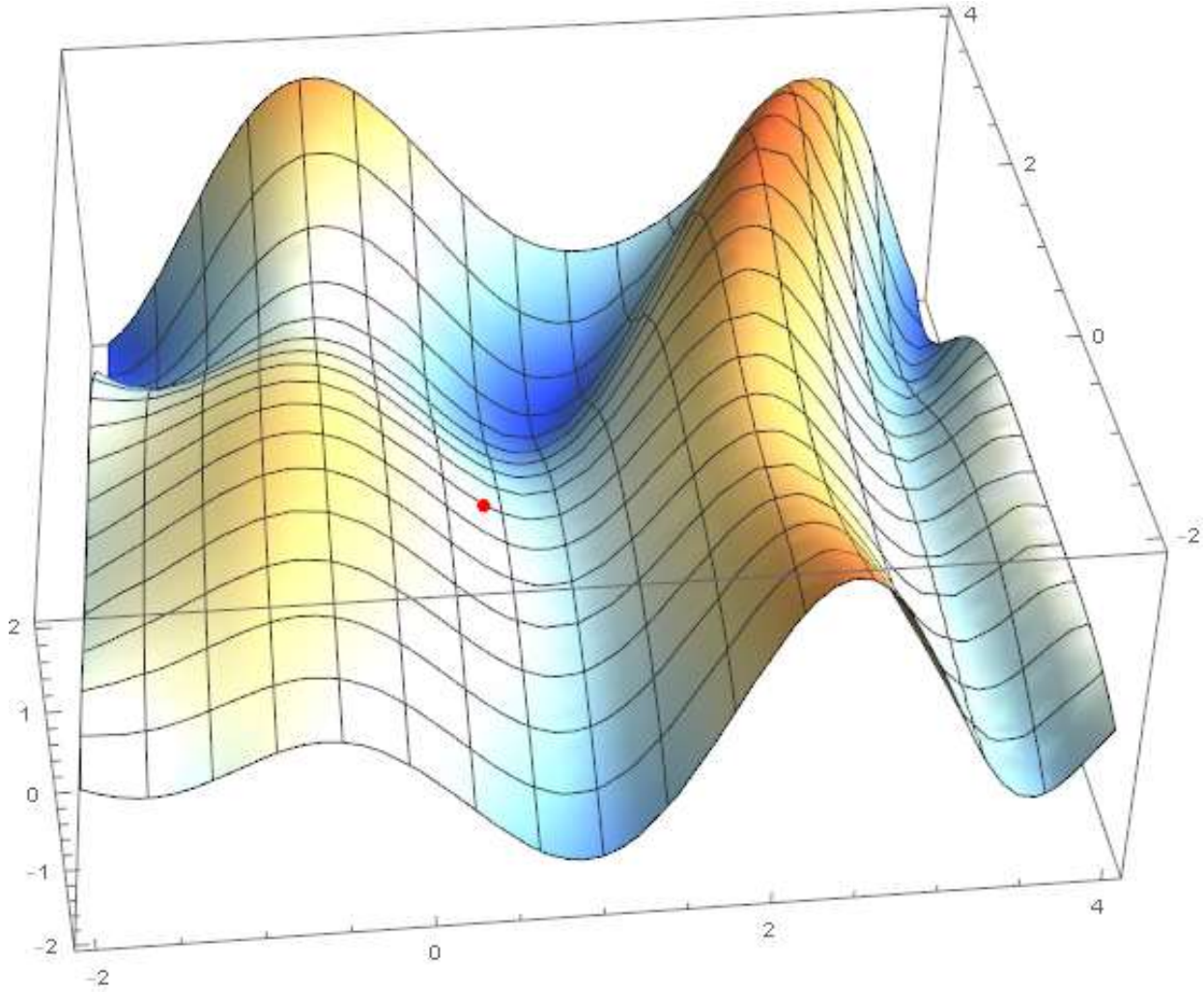

Fig 2.

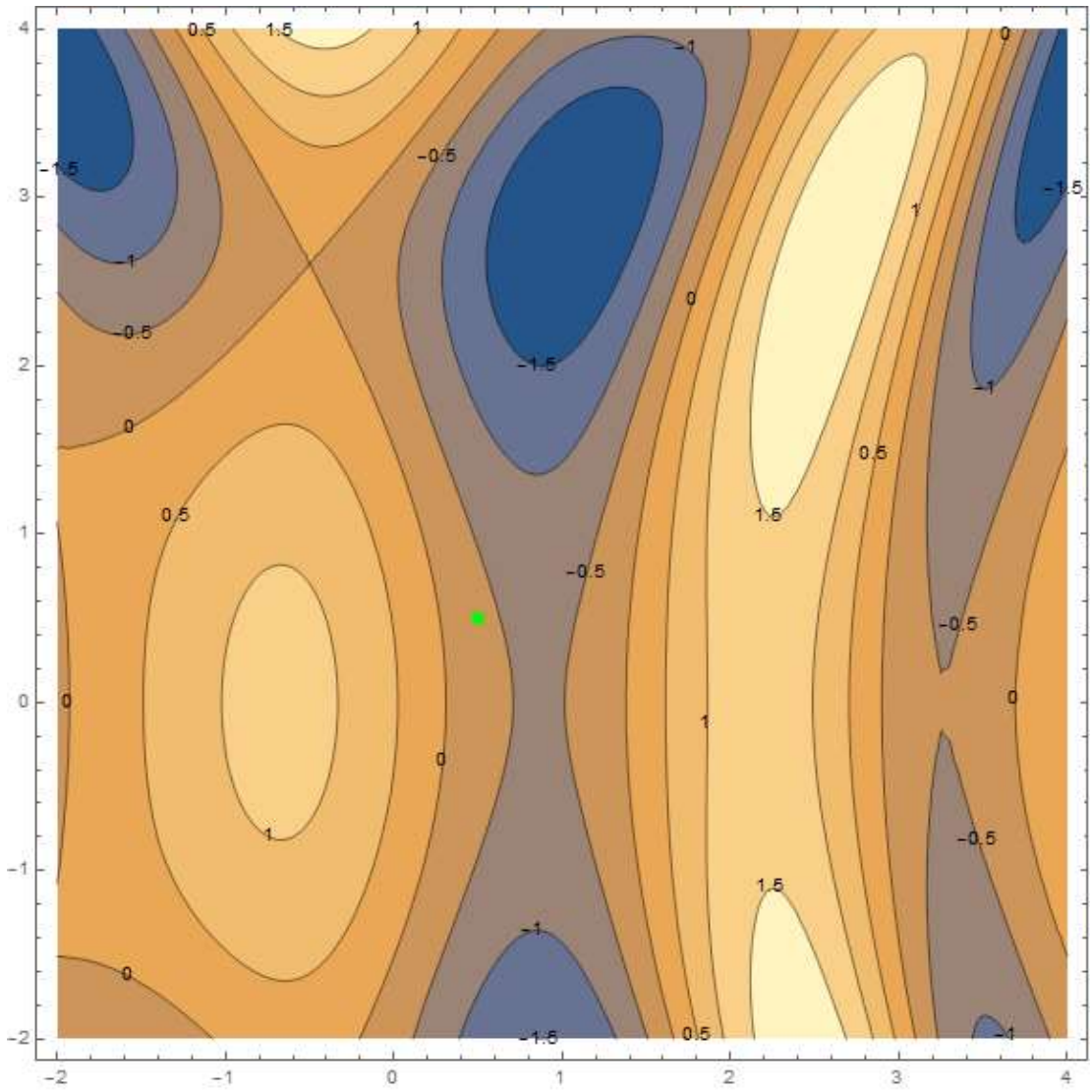

Fig 3.

The essence of the method is to go in the direction opposite to the gradient of the function at the current point. A gradient is a vector that indicates the direction of the greatest growth of a function. Mathematically, this is a vector of derivatives with respect to all arguments –

$$\mathbf{grad}(\boldsymbol{f}) = \Delta \boldsymbol{f} = \left(\frac{\partial \boldsymbol{f}}{\partial \boldsymbol{x}}, \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{y}}\right).$$

The function is taken at random and we do not calculate on it.

Separately it is necessary to say about the speed with which you need to move to a minimum (with regard to the task of machine learning, this will be called the speed of training). To obtain the first results, it will be sufficient for us to choose a fixed speed. However, it is often a good idea to downgrade it during the execution of the algorithm, i.e. Move in smaller and smaller steps.

In the following example, we try to restore the value of the $\boldsymbol{2x-3}$ function in the interval from -2 to 2 to 50 points with normally distributed noise. To teach the model, we will be sets (packets) of 5 points using stochastic gradient descent. Let's go directly to the code.

```
import numpy as np
import tensorflow as tf
%matplotlib inline
```

```
import matplotlib.pyplot as plt

samples = 50 # amount of points
packetSize = 5 # package size
def f(x): return 2*x-3 # The function we are looking for
x_0 = -2.0 # Start of interval
x_l = 2.0 # End of interval
sigma = 0.5 # Root-mean-square deviation of noise

np.random.seed(0) # Make randomness predictable (so that everyone can
repeat the calculations on the same data set)
data_x = np.arange(x_0,x_l,(x_l-x_0)/samples) # Massif [-2, -1.92, -1.
84, ..., 1.92, 2]
np.random.shuffle(data_x) # Mixing
data_y = list(map(f, data_x)) + np.random.normal(0, sigma, samples) #
Array of function values with noise
print(",".join(list(map(str,data_x[:packetSize])))) # The first packag
e x
print(",".join(list(map(str,data y[:packetSize])))) # The first packag
e y

tf_data_x = tf.placeholder(tf.float32, shape=(packetSize,)) # Node on
which we will give arguments to the function
tf_data_y = tf.placeholder(tf.float32, shape=(packetSize,)) # Node on
which we will give the values of the function

weight = tf.Variable(initial_value=0.1, dtype=tf.float32, name="a")
bias = tf.Variable(initial_value=0.0, dtype=tf.float32, name="b")
model = tf.add(tf.add(tf.multiply(tf_data_x1, weight1), tf.multiply(tf
_data_x2, weight2)), bias)

loss = tf.reduce_mean(tf.square(model-tf_data_y)) # Loss function, abo
ut it below
optimizer = tf.train.GradientDescentOptimizer(0.5).minimize(loss) # Me
thod of optimization, it is also lower

with tf.Session() as session:
    tf.global_variables_initializer().run()
    for i in range(samples//packetSize):
        feed_dict={tf_data_x: data_x[i*packetSize:(i+1)*packetSize], t
f_data_y: data_y[i*packetSize:(i+1)*packetSize]}
        _, l = session.run([optimizer, loss], feed_dict=feed_dict) # R
un the optimizer and calculate the "loss"
        print("error: %f" % (l, ))
        print("a = %f, b = %f" % (weight.eval(), bias.eval()))
    plt.plot(data_x, list(map(lambda x: weight.eval()*x+bias.eval(), d
ata_x)), data_x, data_y, 'ro')
```

Conclusion

```
[ 0.24 -1.12 -1.2   1.28 -1.84]
[-2.71939309 -5.65359827 -5.60787235 -0.7022561  -6.27344936]
ошибка: 21.224598
a = 4.555096, b = -4.138514
ошибка: 6.941454
a = 2.484084, b = -2.752421
ошибка: 0.399143
a = 2.223793, b = -3.131071
ошибка: 0.238383
a = 2.134463, b = -2.752734
ошибка: 1.017416
a = 1.630625, b = -3.502803
ошибка: 0.928532
a = 1.964050, b = -2.789906
ошибка: 0.096641
a = 2.248643, b = -3.046779
ошибка: 0.340161
a = 2.146654, b = -2.895813
ошибка: 0.134945
a = 1.902377, b = -2.817182
ошибка: 0.528846
a = 2.276279, b = -3.158037
```

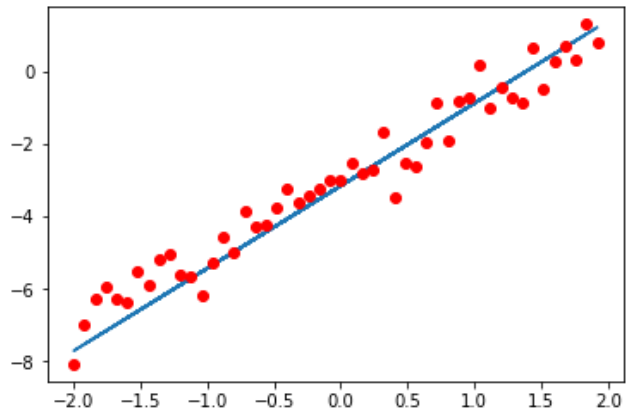

Fig 4.

Our graph looks something like this:

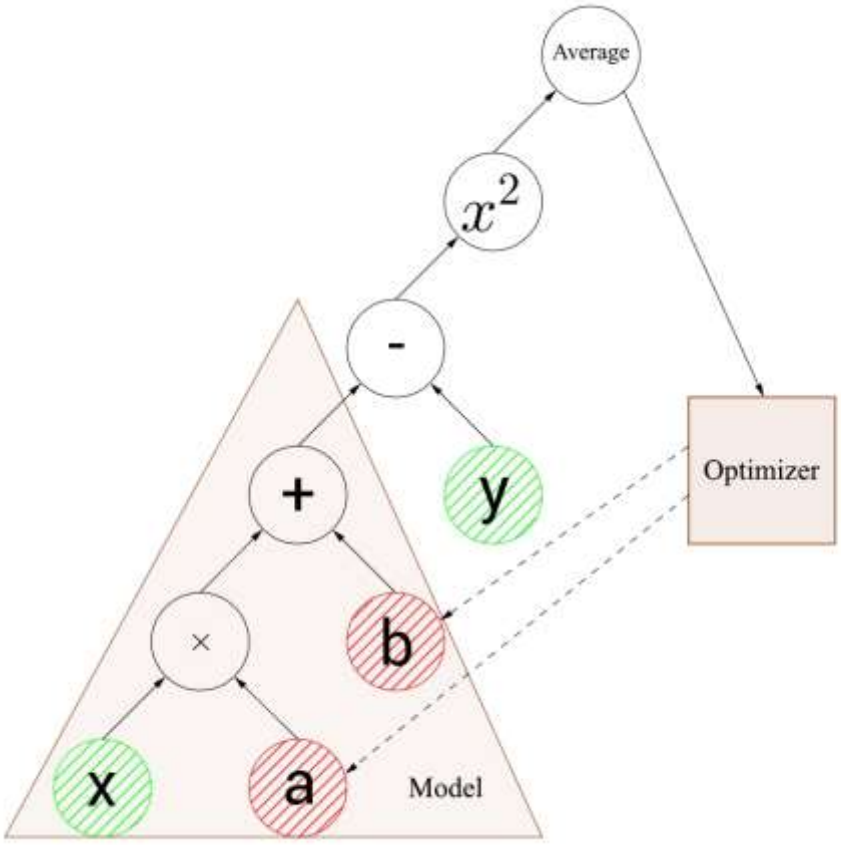

Fig 5. The input nodes are highlighted in green, and the optimizable variables are red.

The first thing that should strike the eye is the mismatch of the dimensions of the input nodes with variables. Input nodes accept arrays of 5 elements, and variables are numbers. This is called broadcasting. Roughly speaking, when you need to perform calculations on arrays, one of which has an extra dimension, the calculations are performed separately for each element of the larger array and the result is an array of larger dimension. That is, $[1,2,3,4,5] + 1 = [2,3,4,5,6]$, it is rather difficult to formulate, but it should be intuitively clear.

Let's manually recalculate the algorithm's actions, I think this is the best way to understand what's going on. So, the arguments are passed to the inputs - $[0.24, -1.12, -1.2, 1.28, -1.84]$ and the values $[-2.72, -5.65, -5.61, -0.70, -6.27]$ (rounded to hundredths). First, we compute the value of the function batch, recall that after initializing variables, the function looks like $0{,}1 \cdot x + 0$. We substitute each argument:

$$\begin{bmatrix} 0.1 \cdot 0.24 + 0 = 0.024 \\ 0.1 \cdot -1.12 + 0 = -0.112 \\ 0.1 \cdot -1.2 + 0 = -0.12 \\ 0.1 \cdot 1.28 + 0 = 0.128 \\ 0.1 \cdot -1.84 + 0 = -0.184 \end{bmatrix}$$

Further, the obtained values are subtracted from the standard values, squared, and the average value is calculated:

$$\begin{bmatrix} (0.024-(-2.72))^2 \approx 7.53 \\ (-0.112-(-5.65))^2 \approx 30.67 \\ (-0.12-(-5.61))^2 \approx 30.14 \\ (0.128-0.7)^2 \approx 0.69 \\ (-0.184-(-6.27))^2 \approx 37.04 \end{bmatrix} \Rightarrow \frac{7.53+30.67+30.14+0.69+37.04}{5} \approx 21.21$$

The difference of about a hundredth of the value deduced to the log is caused by rounding to the nearest hundredths during the calculations. So, we decided to make the mistake, now it's time to deal with optimization. In the graph image above, with dotted arrows, it is shown that the optimizer modifies the variables. In this example, a stochastic gradient descent with a speed of 0.5 is used. Let's take a step, we optimize the variables a and b, so we find the gradient from them:

$$f=(a\cdot x+b-y)^2 \Rightarrow \begin{cases} \frac{\partial f}{\partial a}=2x(ax+b-y) \\ \frac{\partial f}{\partial b}=2(ax+b-y) \end{cases}$$

We need to improve the value over the entire set of points, so calculate the average value of the gradient, for convenience, for each variable separately:

$$a \Rightarrow \frac{1.31712+(-12.4051)+(-13.176)+2.11968+(-22.3965)}{5}=-8.90816$$
$$b \Rightarrow \frac{5.488+11.076+10.98+1.656+12.172}{5}=8.2744$$

Finally, we change the values of the optimized variables taking into account the given speed:

$$a_{new}=a_{old}-0.5\cdot -8.90816=0.1-0.5\cdot(-8.90816)=4.55$$
$$b_{new}=b_{old}-0.5\cdot 8.2744=0-0.5\cdot(-8.90816)=-4.14$$

The values of $\boldsymbol{a_{new}}$ and $\boldsymbol{b_{new}}$ are the required values of the variables. These calculations are repeated in a cycle on each set of points. Why is this method called stochastic? Because we calculate the gradient only on a small piece of data (a package), not at all points at once. Thus, a stochastic descent requires much less computation, but does not guarantee a reduction in error at each iteration. Strange as it may seem, this "noise" in the value of time convergence is even useful, because Allows to "get out" of local minima.