Machine Learning for NLP
Lecture 3: Optimization and machine learning

UNIVERSITY OF
GOTHENBURG

Richard Johansson

September 8, 2016

linear classifiers

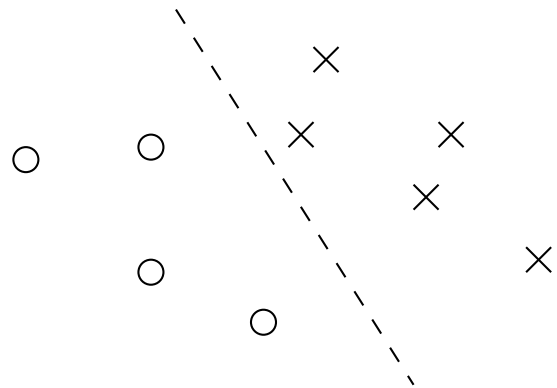
- a **linear classifier** is a classifier that is defined in terms of a scoring function like this

$$\text{score} = \boldsymbol{w} \cdot \boldsymbol{x}$$

- explanation of the parts:
  - $\boldsymbol{x}$ is a vector with features of what we want to classify (e.g. made with a `DictVectorizer`)
  - $\boldsymbol{w}$ is a vector representing which features the classifier thinks are important
  - $\cdot$ is the dot product between the two vectors
- for now, we'll assume that there are two classes: **binary** classification
  - return the first class if the score > 0
  - . . . otherwise the second class
- the essential idea: **features are scored independently**

geometric view

- geometrically, a linear classifier can be interpreted as separating the vector space into two regions with a line (plane, hyperplane)



optimization and machine learning

- we will now consider models that are less ad-hoc than the perceptron
- idea: define an **objective function** based on the fundamental tradeoff in machine learning:
  - how well we handle the training set (**loss**)
  - simplicity of the model (**regularization**)
- . . . and then the training consists of applying **optimization** techniques to find the best $\boldsymbol{w}$
- we will consider two models:
  - logistic regression, based on probability
  - support vector classifier, based on geometry

in scikit-learn

- LR is called `sklearn.linear_model.LogisticRegression`
- SVM is called `sklearn.svm.LinearSVC`

overview

logistic regression

support vector machines

basic optimization

practical information

linear classifiers with probabilities?

- linear classifiers select the outputs based on a scoring function:

$$\text{score} = \boldsymbol{w} \cdot \boldsymbol{x}$$

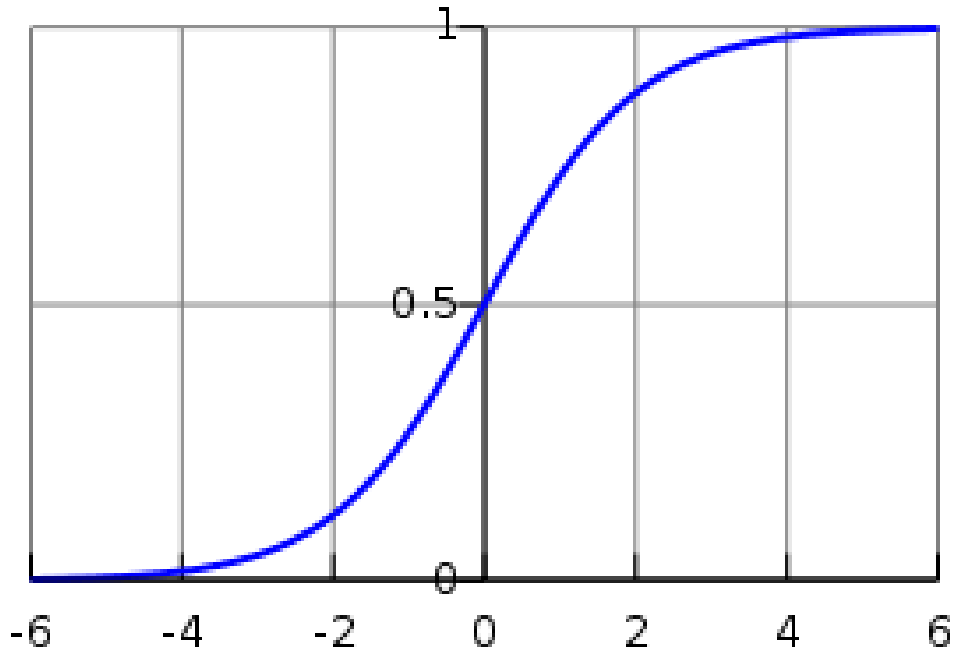
- how to convert the scores into probabilities?
- idea: use a **logistic** or **sigmoid** function:

$$P(\text{positive output}|\boldsymbol{x}) = \frac{1}{1 + e^{-\text{score}}}$$

where $e^{-\text{score}} = \texttt{math.exp(-score)}$

- this is formally a probability: always between 0 and 1, sum of probablities of possible outcomes = 1

the logistic function



conversely

$$P(\text{negative output}|\boldsymbol{x}) = \frac{1}{1 + e^{\text{score}}}$$

making it a bit more compact

- if we code the positive class as +1 and the negative class as -1, then we can write the probability a bit more neatly:

$$P(y|\boldsymbol{x}) = \frac{1}{1 + e^{-y \cdot \text{score}}}$$

recall: the maximum likelihood principle

- select the model that gives a high probability to the data
- in our case, the "model" is the weight vector $\boldsymbol{w}$
- adjust $\boldsymbol{w}$ so that each output label gets a high probability

the likelihood function

- formally, the "probability of the data" is defined by the **likelihood function**
- this is the product of the probabilities of all m individual training instances:

$$L(\boldsymbol{w}) = P(y_1|\boldsymbol{x}_1) \cdot \ldots \cdot P(y_T|\boldsymbol{x}_m)$$

- in our case, this means

$$L(\boldsymbol{w}) = \frac{1}{1 + e^{-y_1 \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_1)}} \cdot \ldots \cdot \frac{1}{1 + e^{-y_m \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_m)}}$$

rewriting a bit...

- we rewrite the previous formula

$$L(\boldsymbol{w}) = \frac{1}{1 + e^{-y_1 \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_1)}} \cdot \ldots \cdot \frac{1}{1 + e^{-y_m \cdot (\boldsymbol{w} \cdot \boldsymbol{x}_m)}}$$

- as

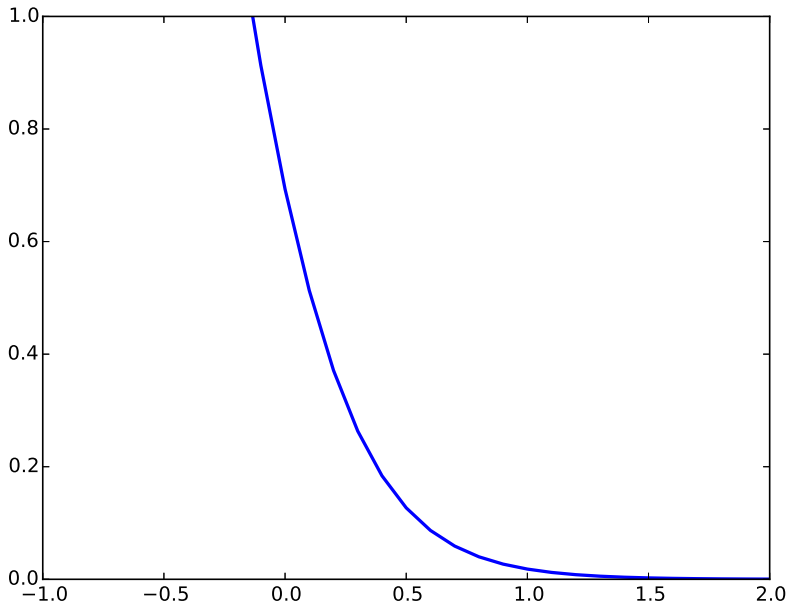
$$-\log L(\boldsymbol{w}) = \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_1, y_1) + \ldots + \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_m, y_m)$$

where

$$\text{Loss}(\boldsymbol{w}, \boldsymbol{x}, y) = \log(1 + \exp(-y \cdot (\boldsymbol{w} \cdot \boldsymbol{x})))$$

is called the **log loss** function

plot of the log loss



recall: the fundamental tradeoff in machine learning

- **goodness of fit**: the learned classifier should be able to correctly classify the examples in the training data
- **regularization**: the classifier should be simple
- but so far in our LR description, we've just taken care of the first part!

what does it mean to "keep the classifier simple"?

- concretely, how can we add regularization to the LR model?
- the most common approach is to add a term that **keeps the weights small**
- formally, we say that the the squared length (norm) of the weight vector should be small:

$$|\boldsymbol{w}|^2 = w_1 \cdot w_1 + \ldots + w_n \cdot w_n = \boldsymbol{w} \cdot \boldsymbol{w}$$

combining the pieces

- we combine the loss and the regularizer:

$$\sum \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \lambda \cdot |\boldsymbol{w}|^2$$

- in this formula, λ is a "tweaking" parameter that controls the tradeoff between loss and regularization
- note: in some formulations (including scikit-learn), there is a parameter C instead of the λ that is put before the loss

this still doesn't look implementable...

- we have an objective function that we want to minimize:

$$\sum \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \lambda \cdot |\boldsymbol{w}|^2$$

- but we still don't know how to convert this into code!

overview

logistic regression

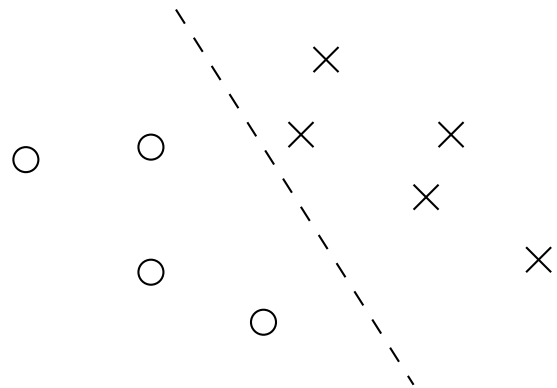
support vector machines

basic optimization

practical information

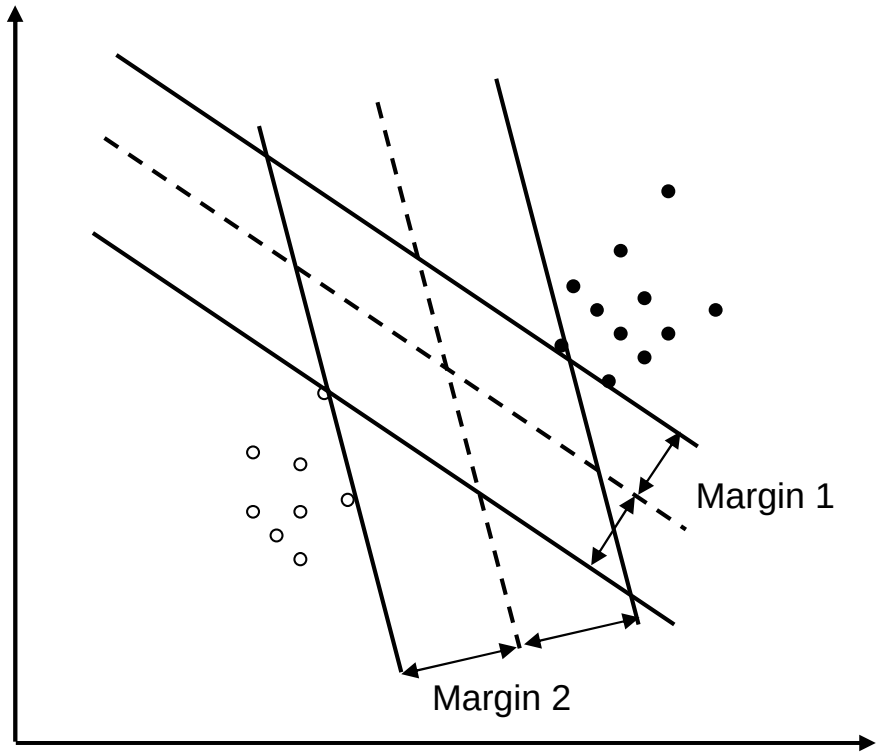
geometric view

- geometrically, a linear classifier can be interpreted as separating the vector space into two regions with a line (plane, hyperplane)

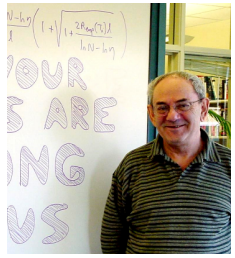


margin of separation

- the **margin** γ denotes how well $\boldsymbol{w}$ separates the classes:



large margins are good



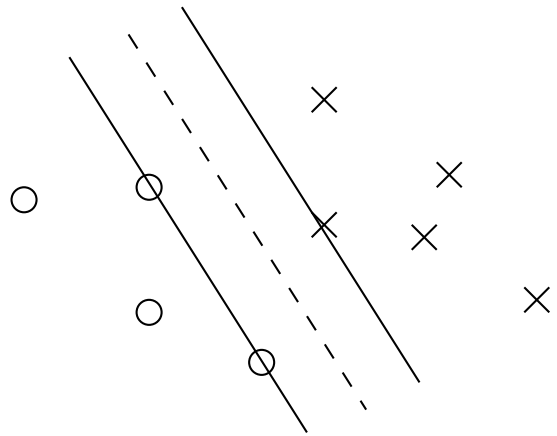
- a result from statistical learning theory:

$$\text{expected test error} = \text{training error} + \text{BigUglyFormula}(\frac{1}{\gamma^2})$$

- larger margin $\rightarrow$ better generalization

support vector machines

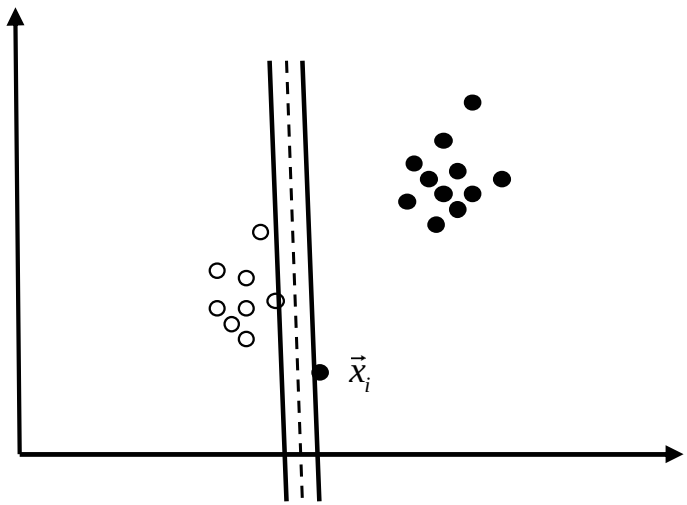
- **support vector machines** (SVMs) or support vector classifiers (SVC) are linear classifiers constructed by selecting the $\boldsymbol{w}$ that maximizes the margin



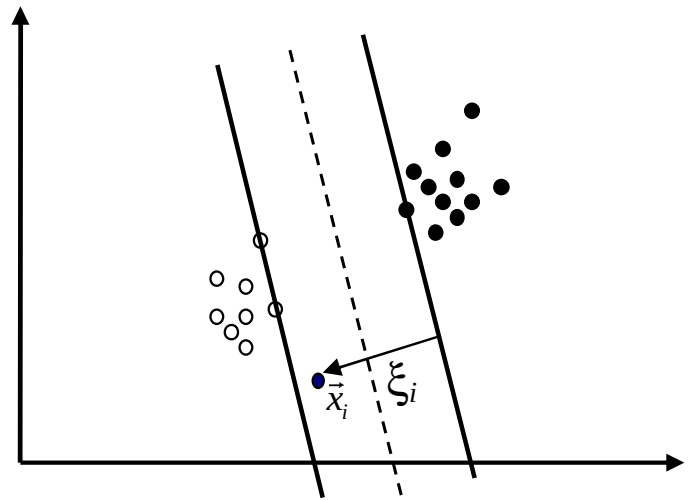
- note: the solution depends only on the borderline examples: the **support vectors**

soft-margin SVMs

- in some cases the dataset is inseparable, or nearly inseparable
- **soft-margin SVM**: allow some examples to be disregarded when maximizing the margin



A) Hard Margin SVM



B) Soft Margin SVM

stating the SVM as using an objective function

- the hard-margin and soft-margin SVM can be stated mathematically in a number of ways
- we'll skip the details, but with a bit of work we can show that the soft-margin SVM can be stated as minimizing

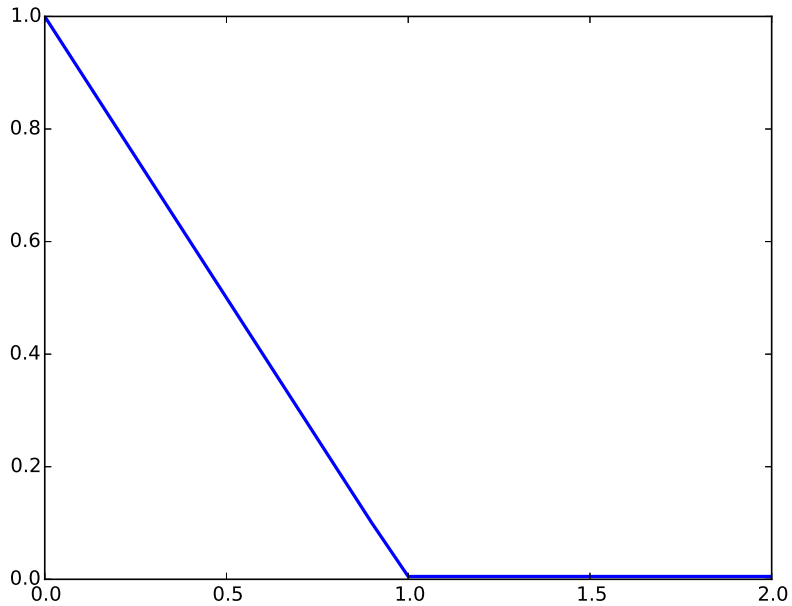
$$\sum \text{Loss}(\boldsymbol{w}, \boldsymbol{x}_i, y_i) + \lambda \cdot |\boldsymbol{w}|^2$$

where

$$\text{Loss}(\boldsymbol{w}, \boldsymbol{x}, y) = \max(0, 1 - y \cdot (\boldsymbol{w} \cdot \boldsymbol{x}))$$

is called the **hinge loss**

plot of the hinge loss



overview

logistic regression

support vector machines

basic optimization

practical information

optimization

- what is optimization?
- **unconstrained optimization**: find the x that gives us the minimal (or maximal) value of some function f:

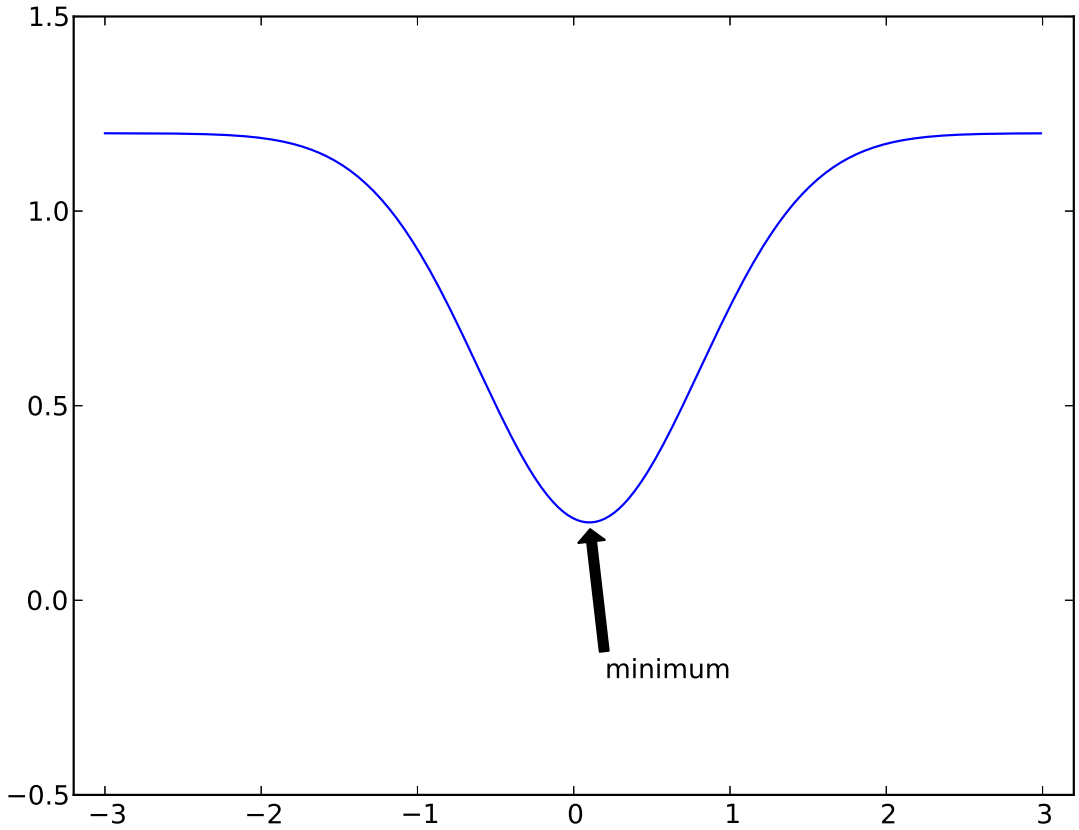
$$\min_x f(x)$$

- **constrained optimization**: find the x that gives us the minimal (or maximal) value of f, where x satisfies some extra conditions:

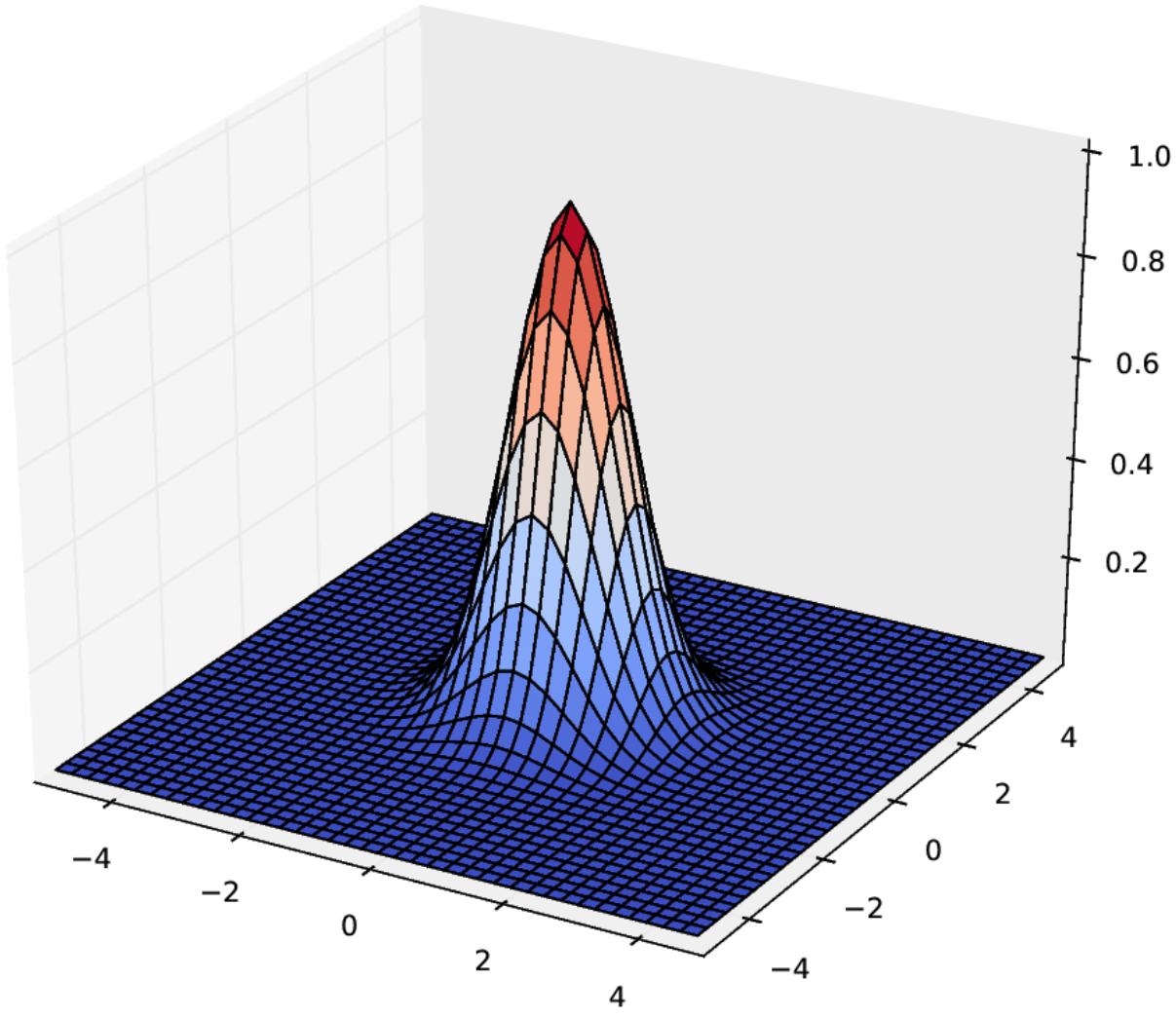
$$\min_x f(x)$$
$$\text{such that } x > 0$$

- today unconstrained optimization only

one-variable example

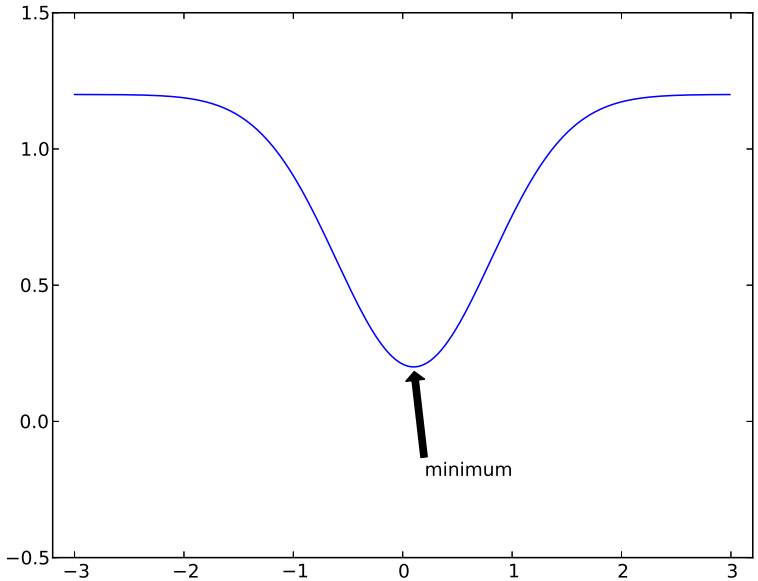


two-variable example



remember your highschool calculus. . .

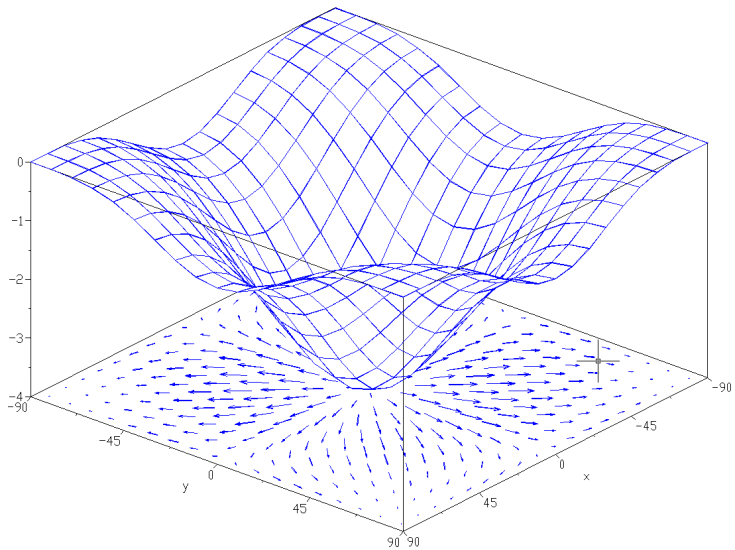
- in your early school days, you might have seen the **derivative** of a function
- intuition: the derivative measures the slope



- if a "nice" function has a maximum or minimum, then the derivative will be zero there

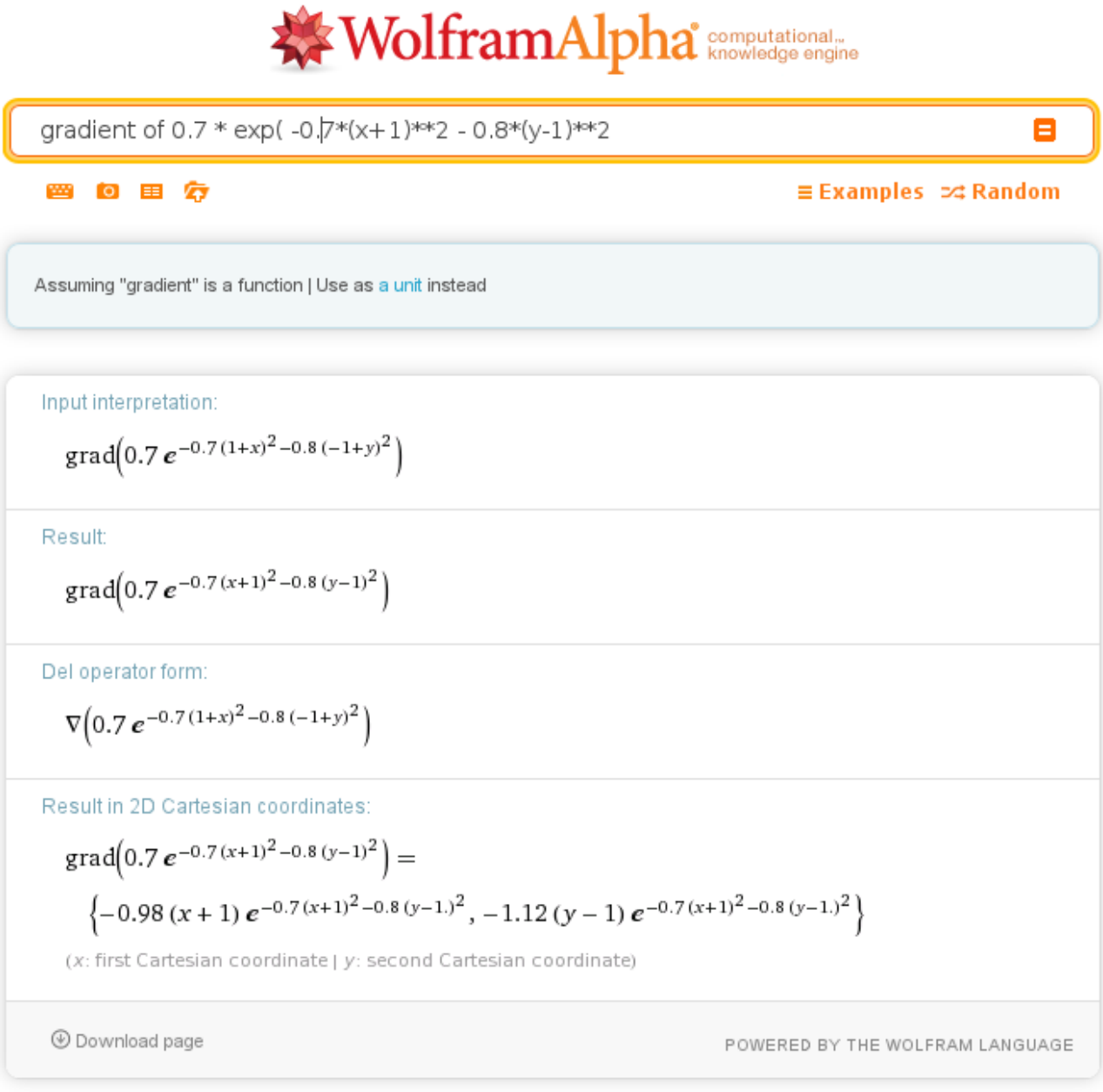
the gradient

- the multidimensional equivalent of the derivative is called the **gradient**
- if f is a function of n variables, then the gradient is an n-dimensional vector, often written $\nabla f(x)$
- intuition: the gradient points in the uphill direction



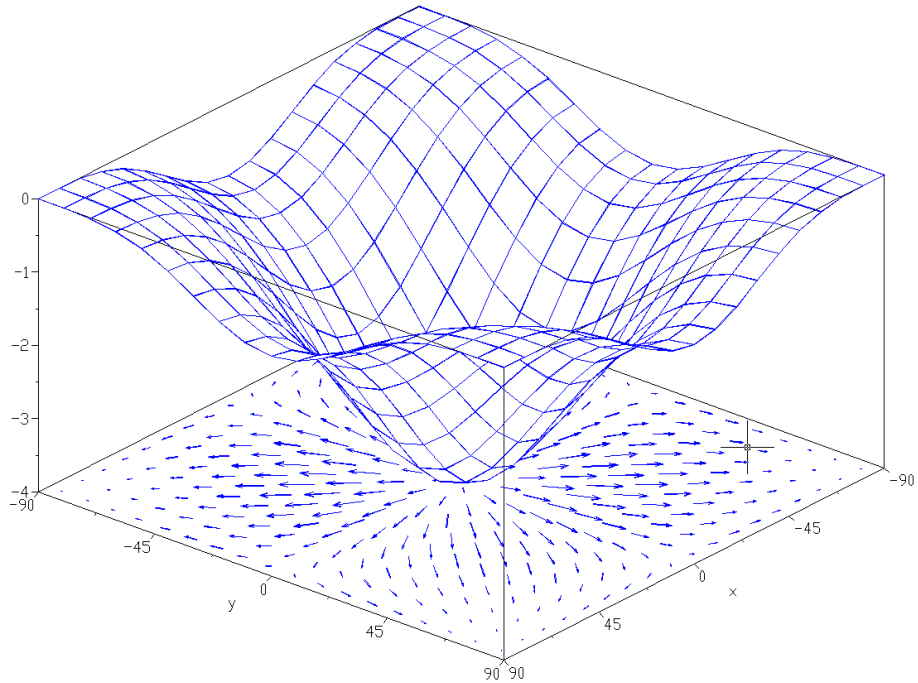
- again: the gradient is zero if we have an optimum

computing the gradient



gradient descent

- as we saw, the gradient points in the uphill direction:



- this intuition leads to a simple idea for finding the minimum:
  - take a small step in the direction opposite to the gradient
  - repeat until the gradient is close enough to zero
- this is called **gradient descent**

gradient descent, pseudocode

- the same thing again, in pseudocode:
  1. set x to some initial value, and select a suitable step size c
  2. compute the gradient $\nabla f(x)$
  3. if $\nabla f(x)$ is small enough, we are done
  4. otherwise, subtract $c \cdot \nabla f(x)$ from x and go back to step 2
- conversely, to find the maximum we can do **gradient ascent**: then we instead add $c \cdot \nabla f(x)$ to x

in Python

```
def gradient_ascent(x_init, y_init,
                    threshold = 0.001,
                    steplength = 0.01):
    x = x_init
    y = y_init
    done = False
    while not done:
        gxy = gradient_of_my_function(x, y)
        x += steplength * gxy[0]
        y += steplength * gxy[1]
        if numpy.linalg.norm(gxy) < threshold:
            done = True
    return (x, y)
```

gradient ascent example

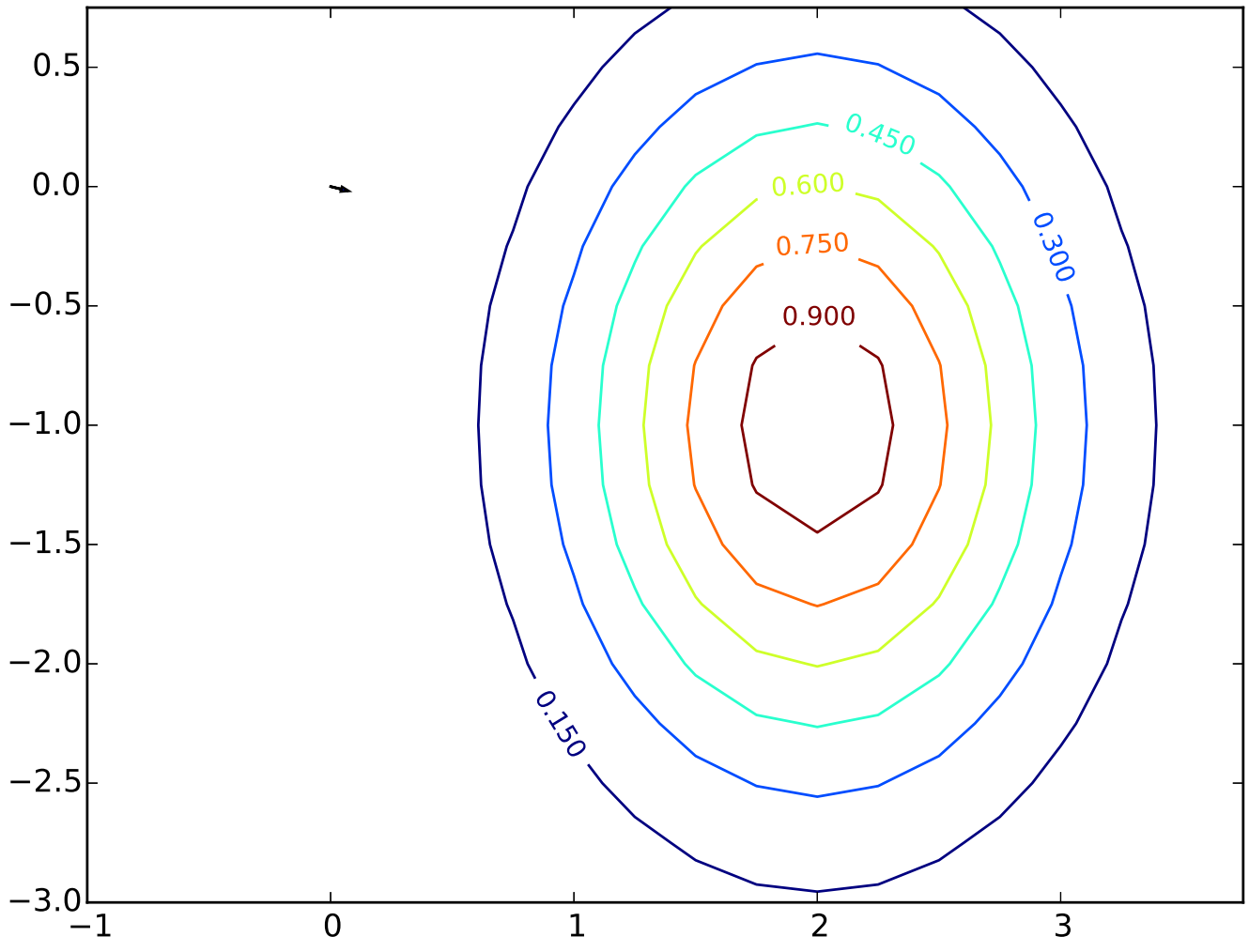
- let's optimize this function:

```
def f(x, y):
    return math.exp(-(x-2)**2 - (y+1)**2)
```

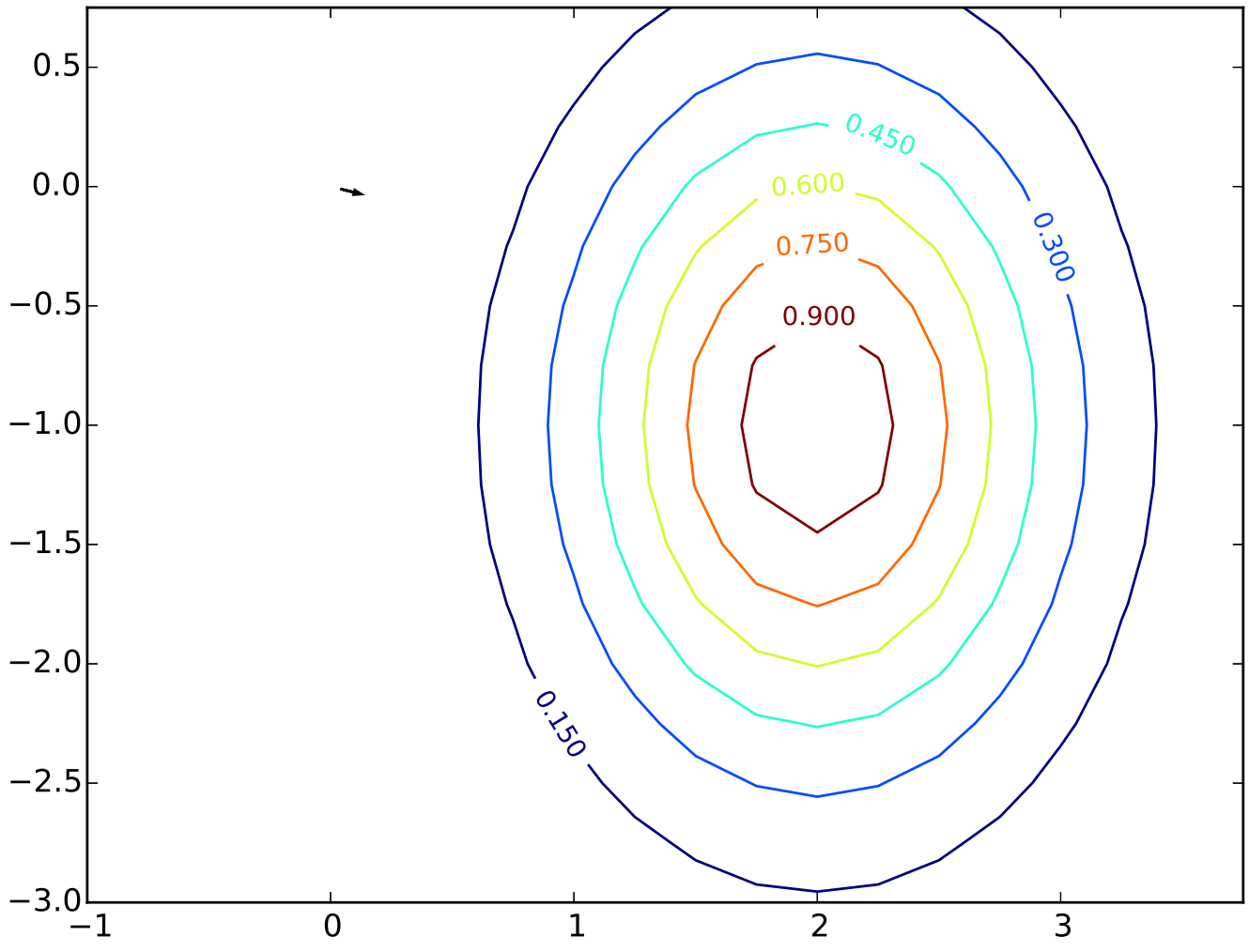
- its gradient is

```
def gradient_of_f(x, y):
    return (-2*(x-2)*f(x, y), -2*(y+1)*f(x, y))
```

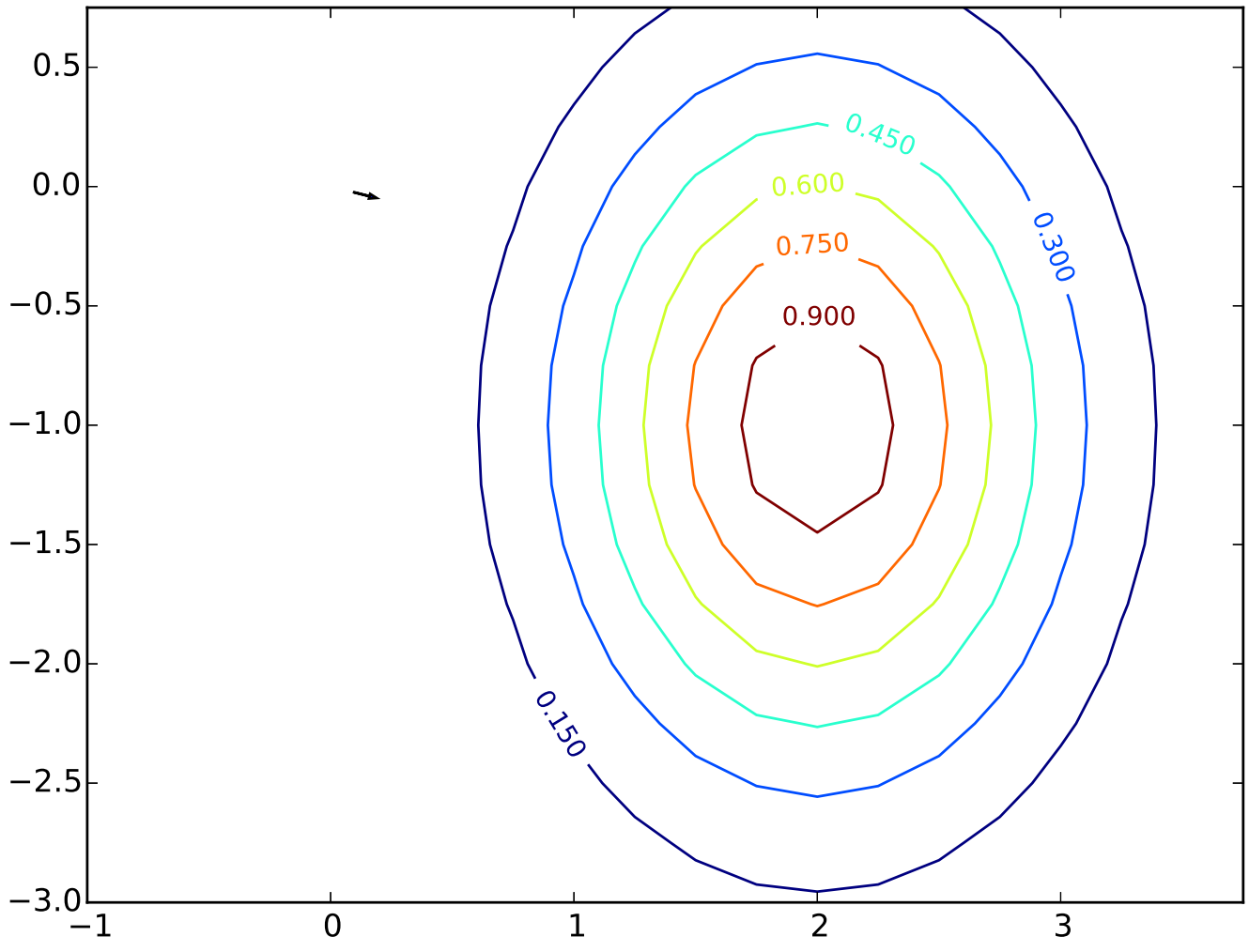
gradient ascent example



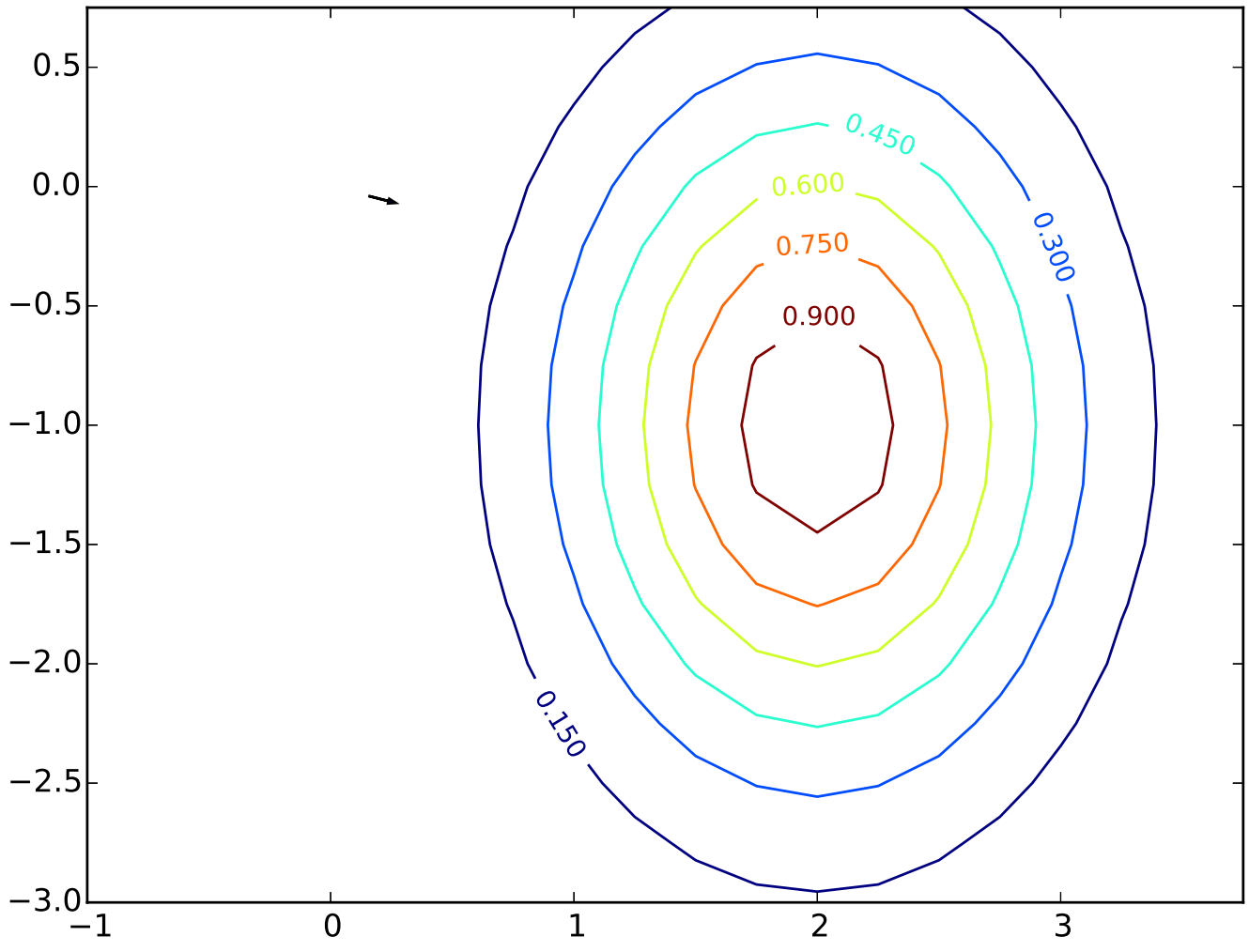
gradient ascent example



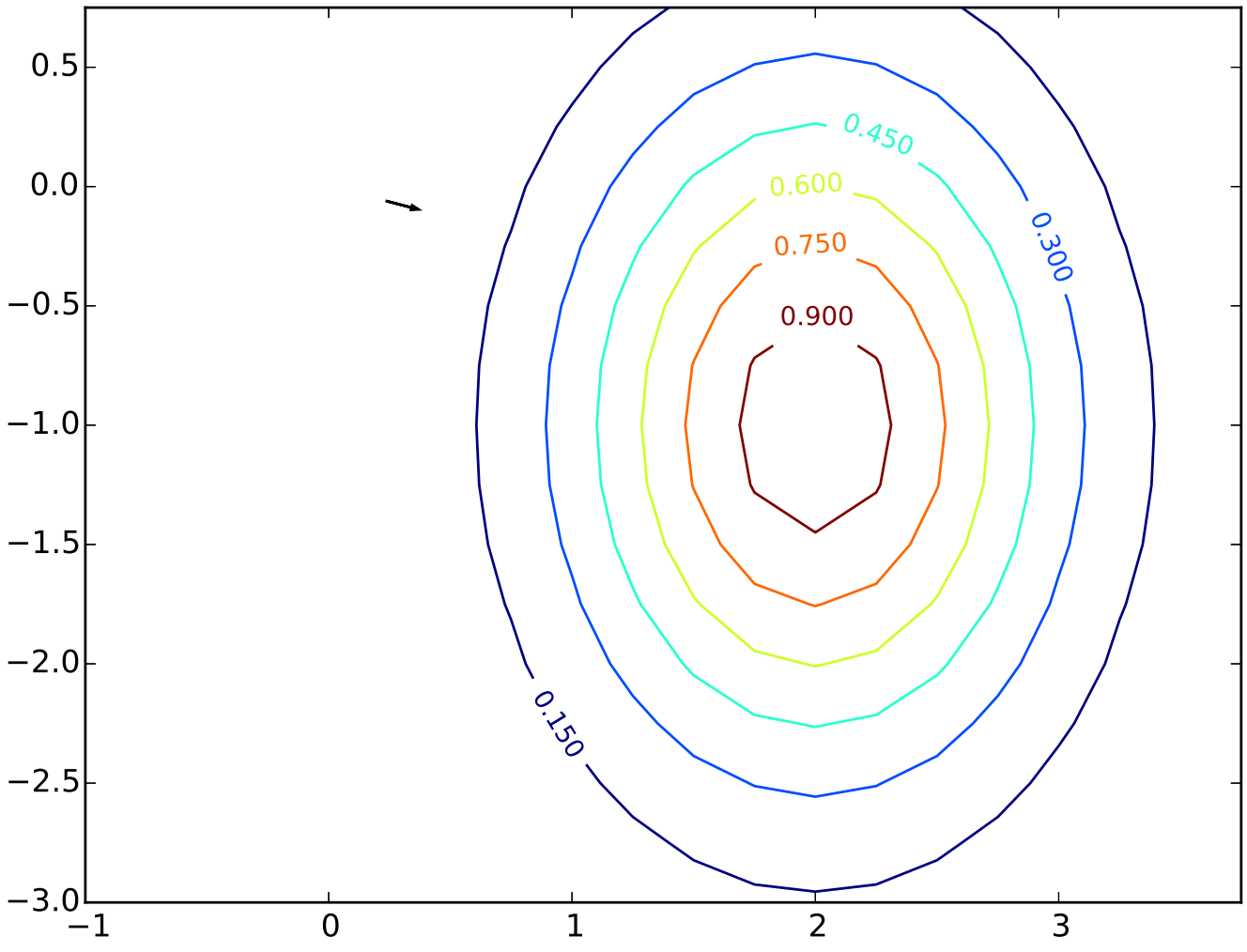
gradient ascent example



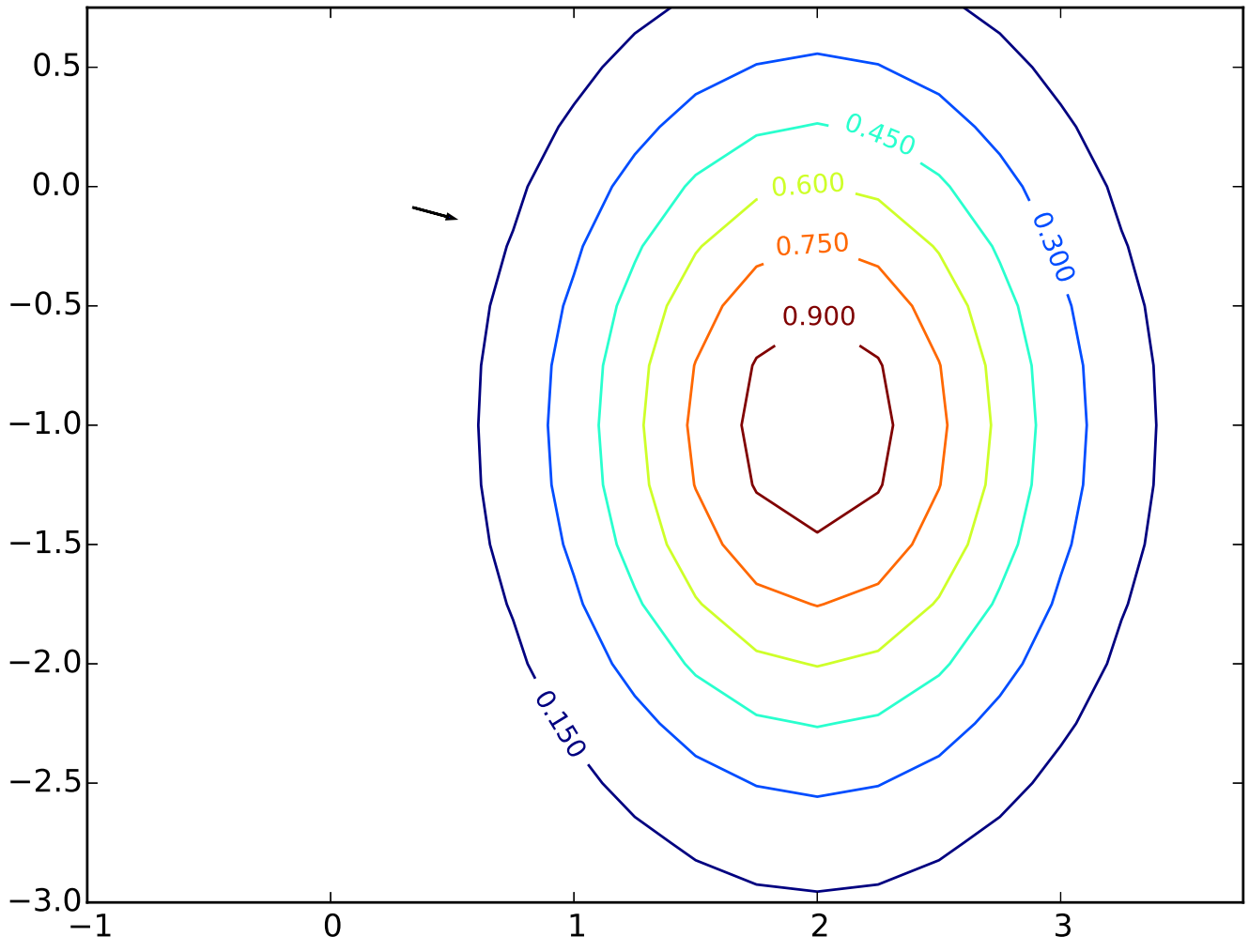
gradient ascent example



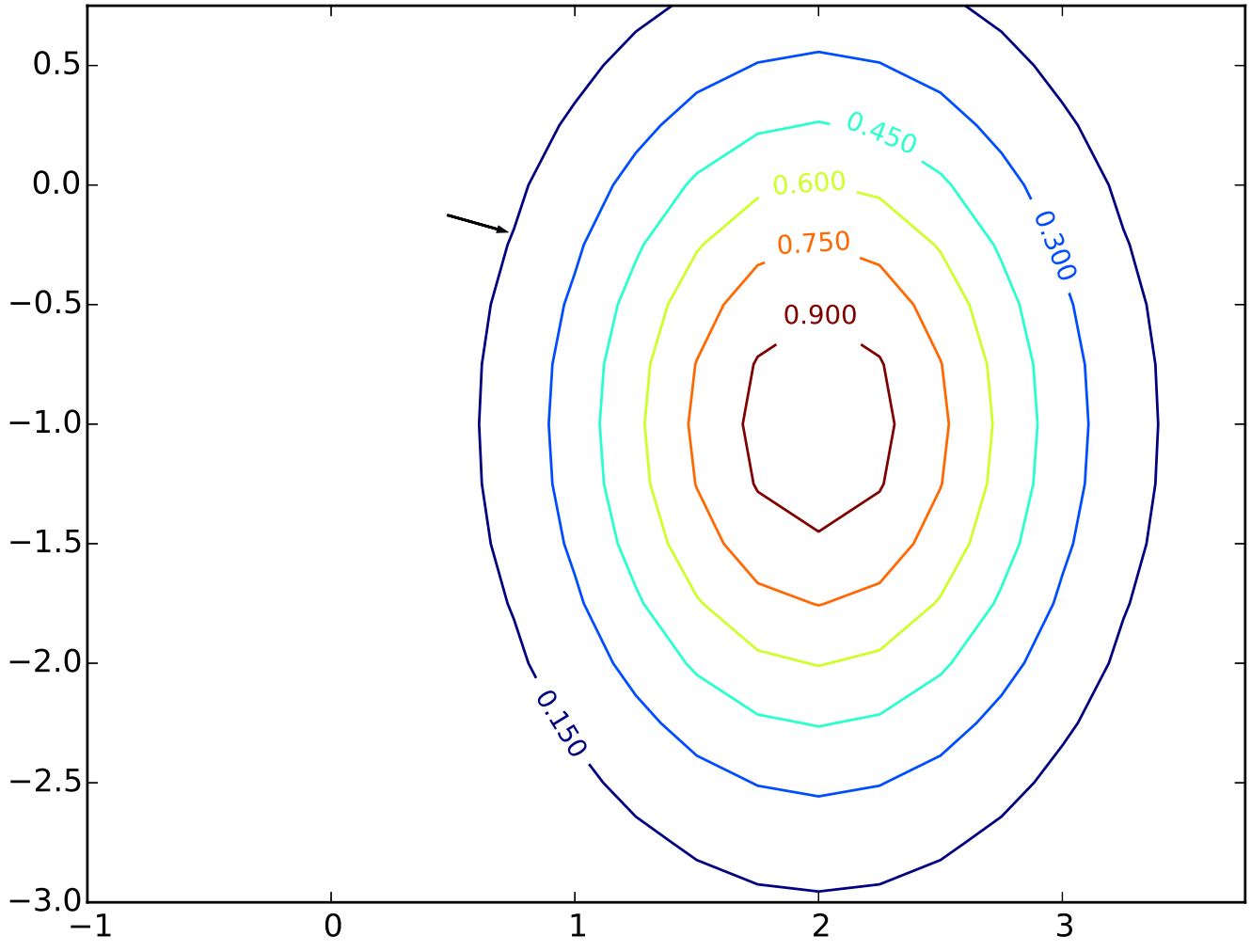
gradient ascent example



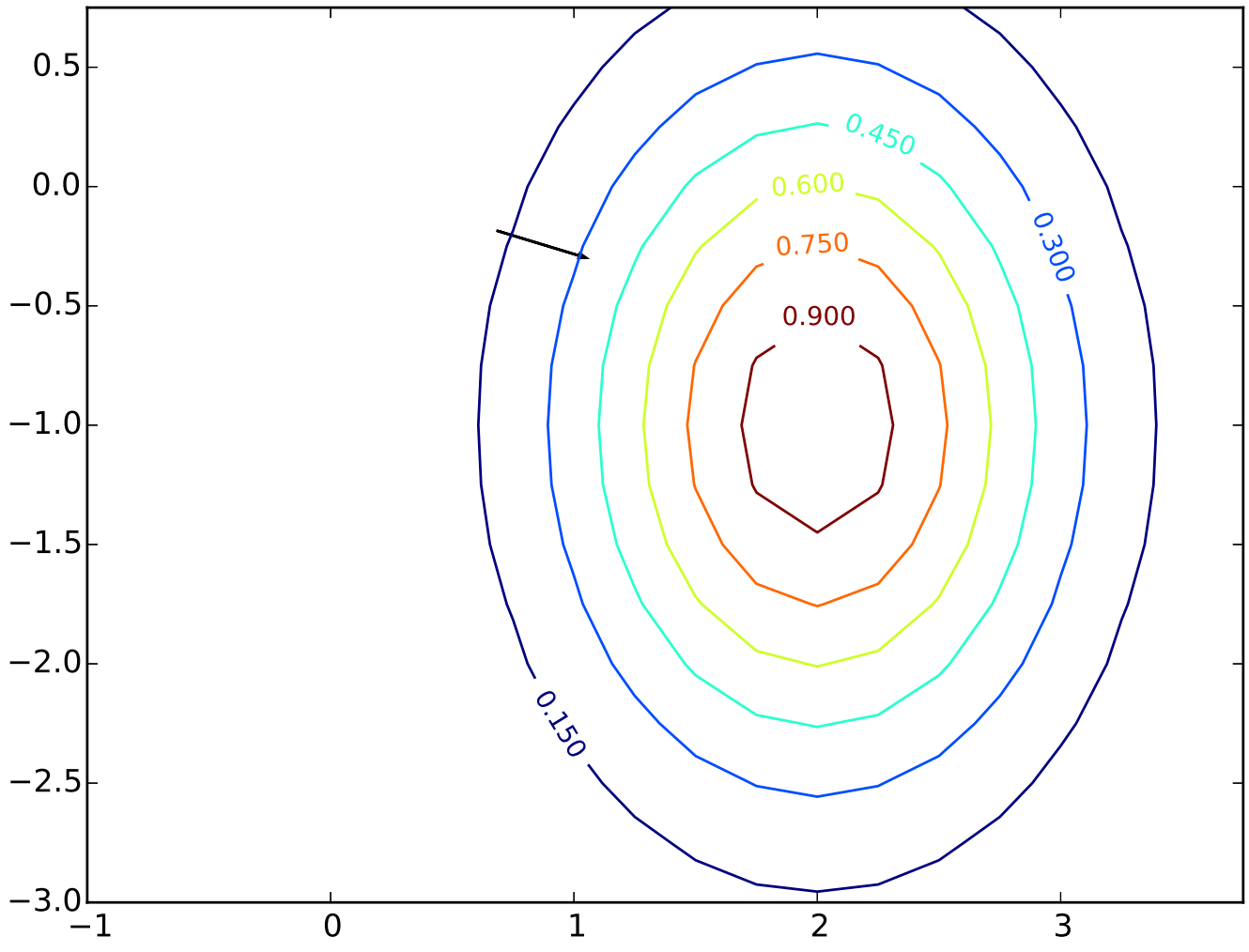
gradient ascent example



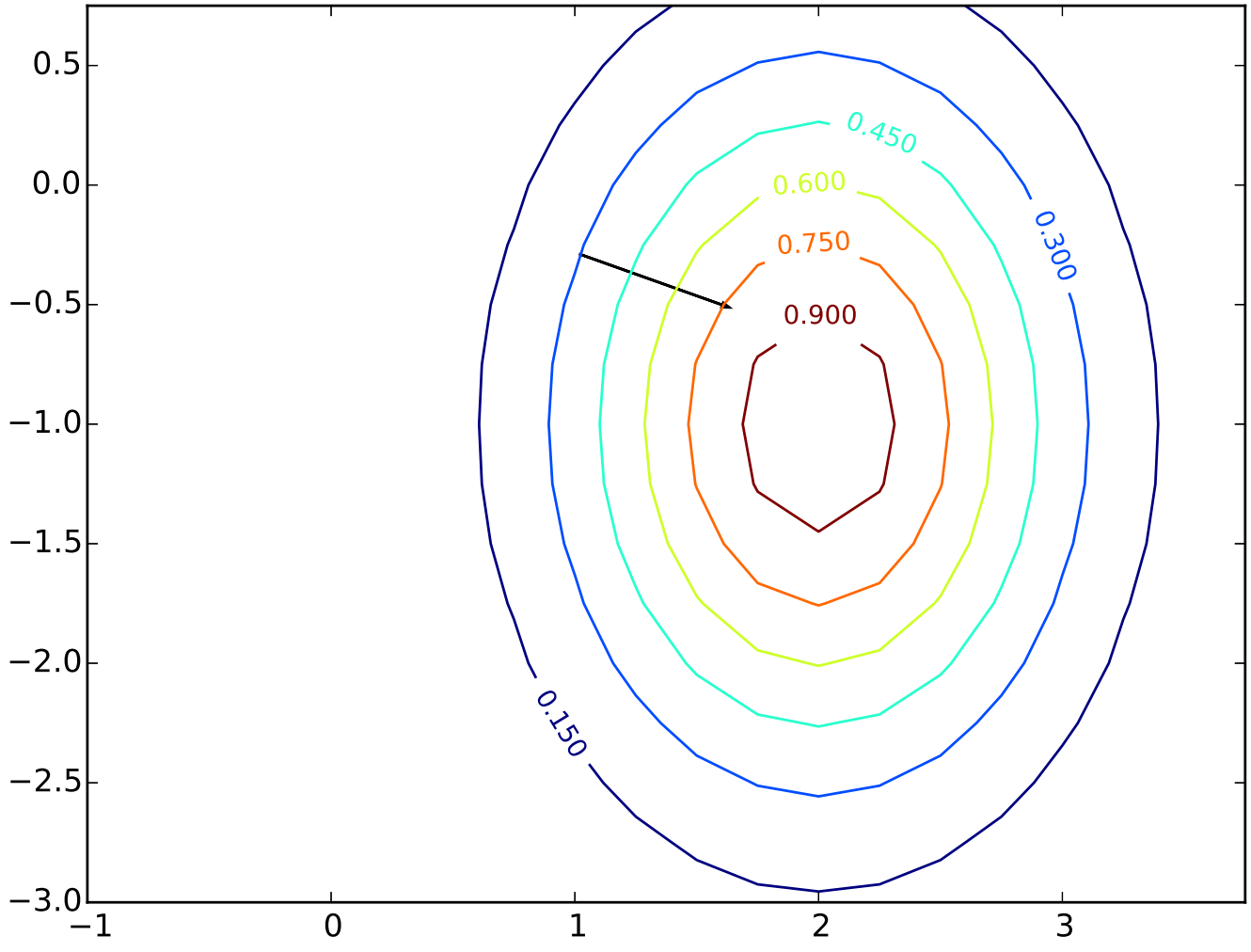
gradient ascent example



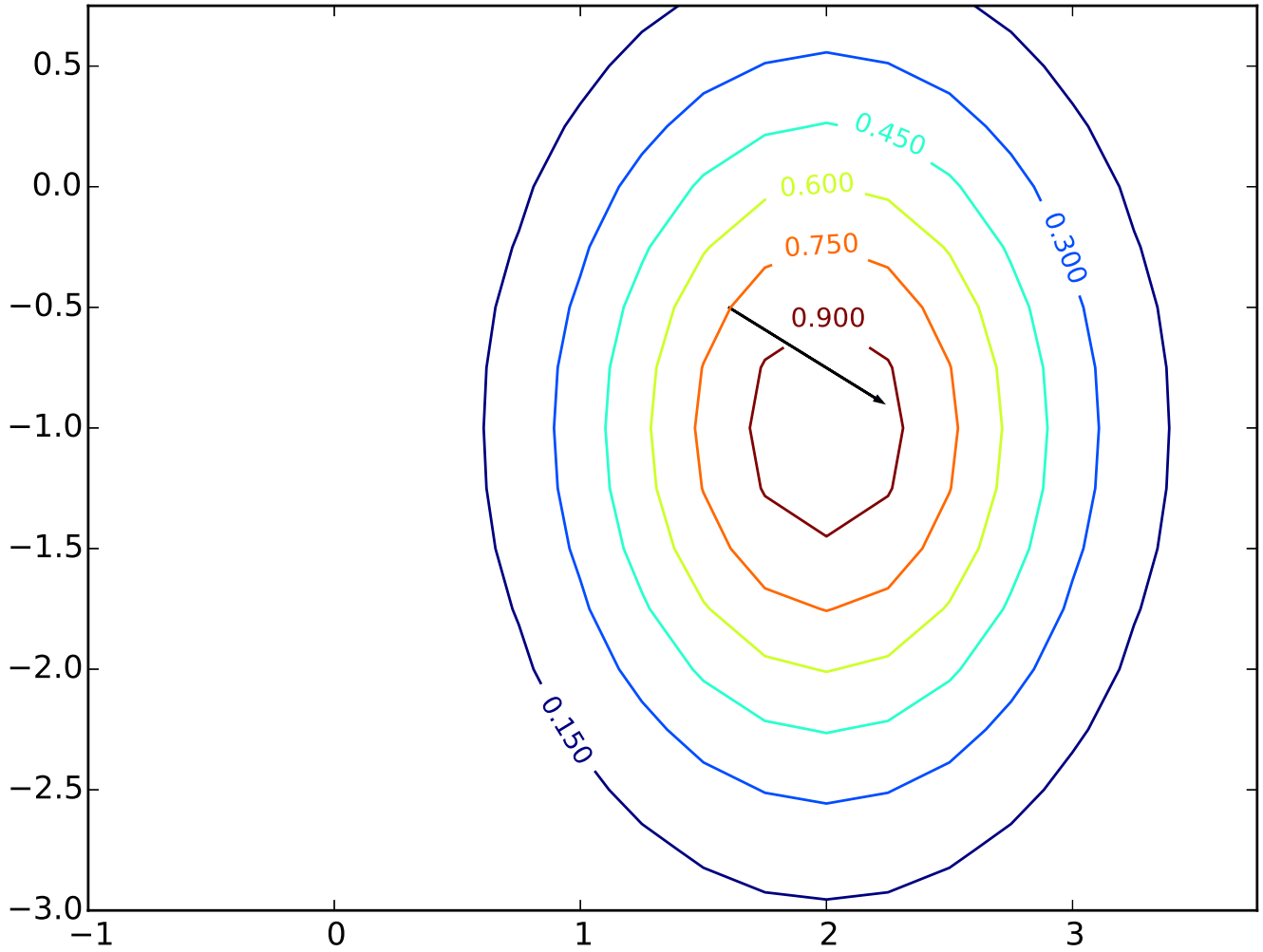
gradient ascent example



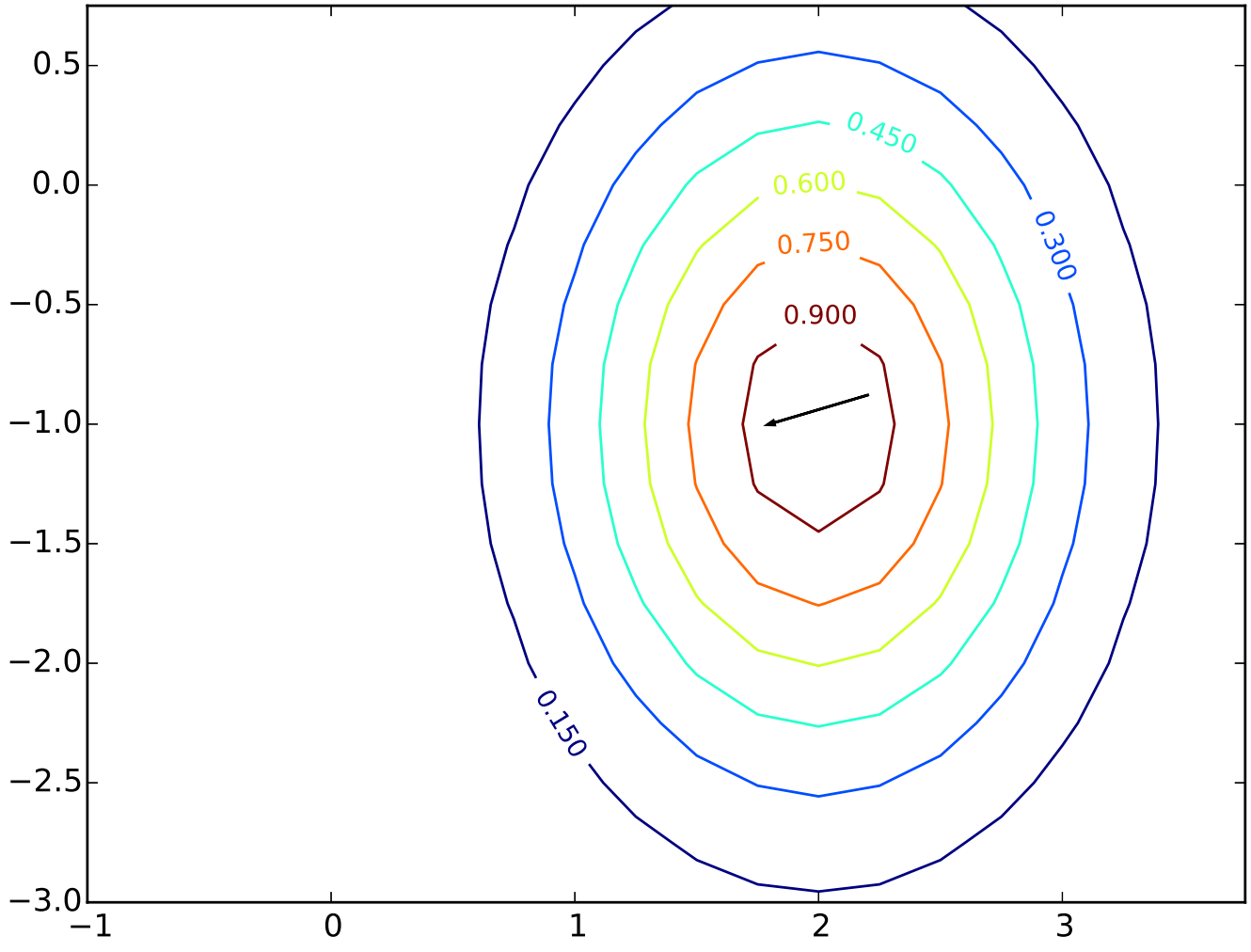
gradient ascent example



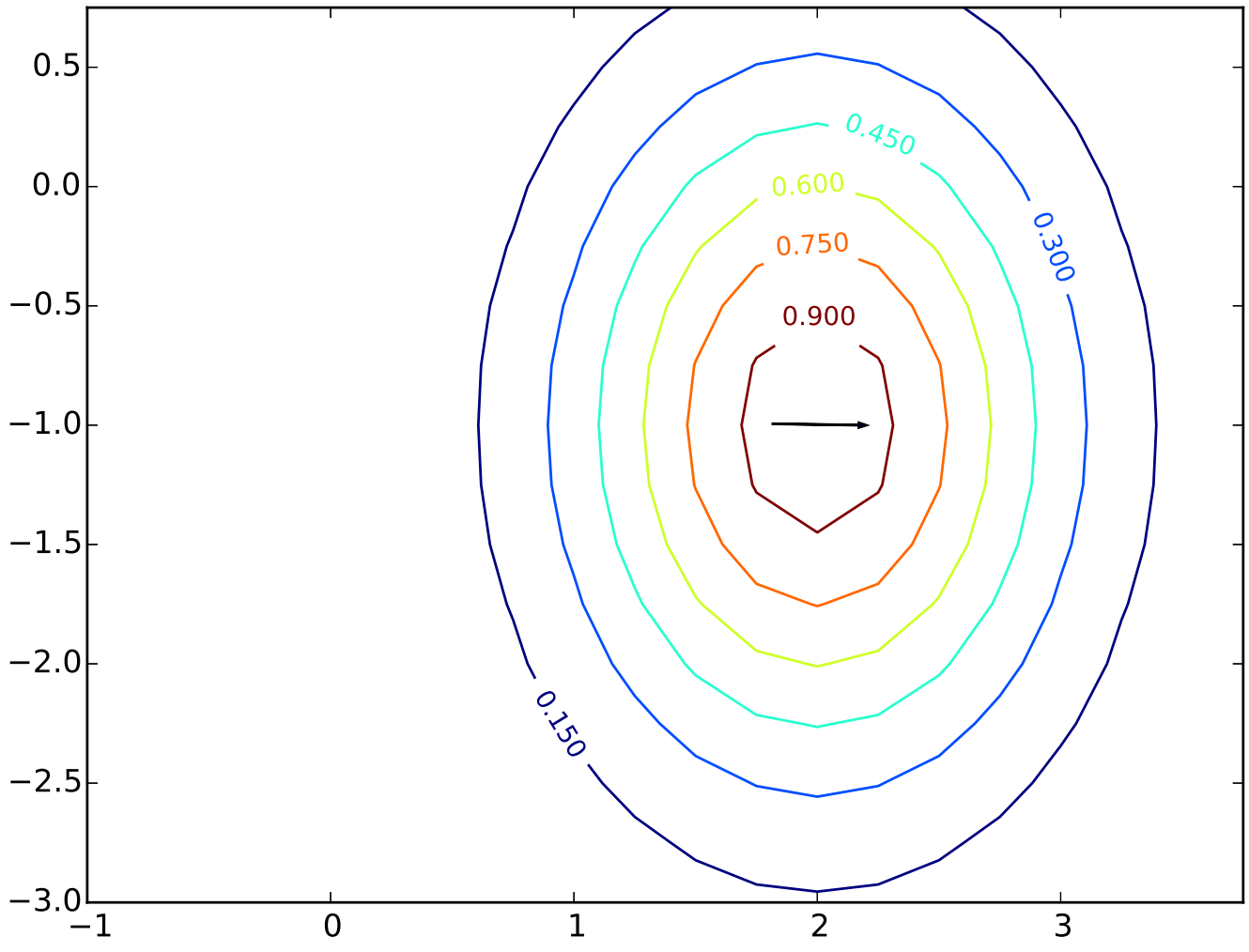
gradient ascent example



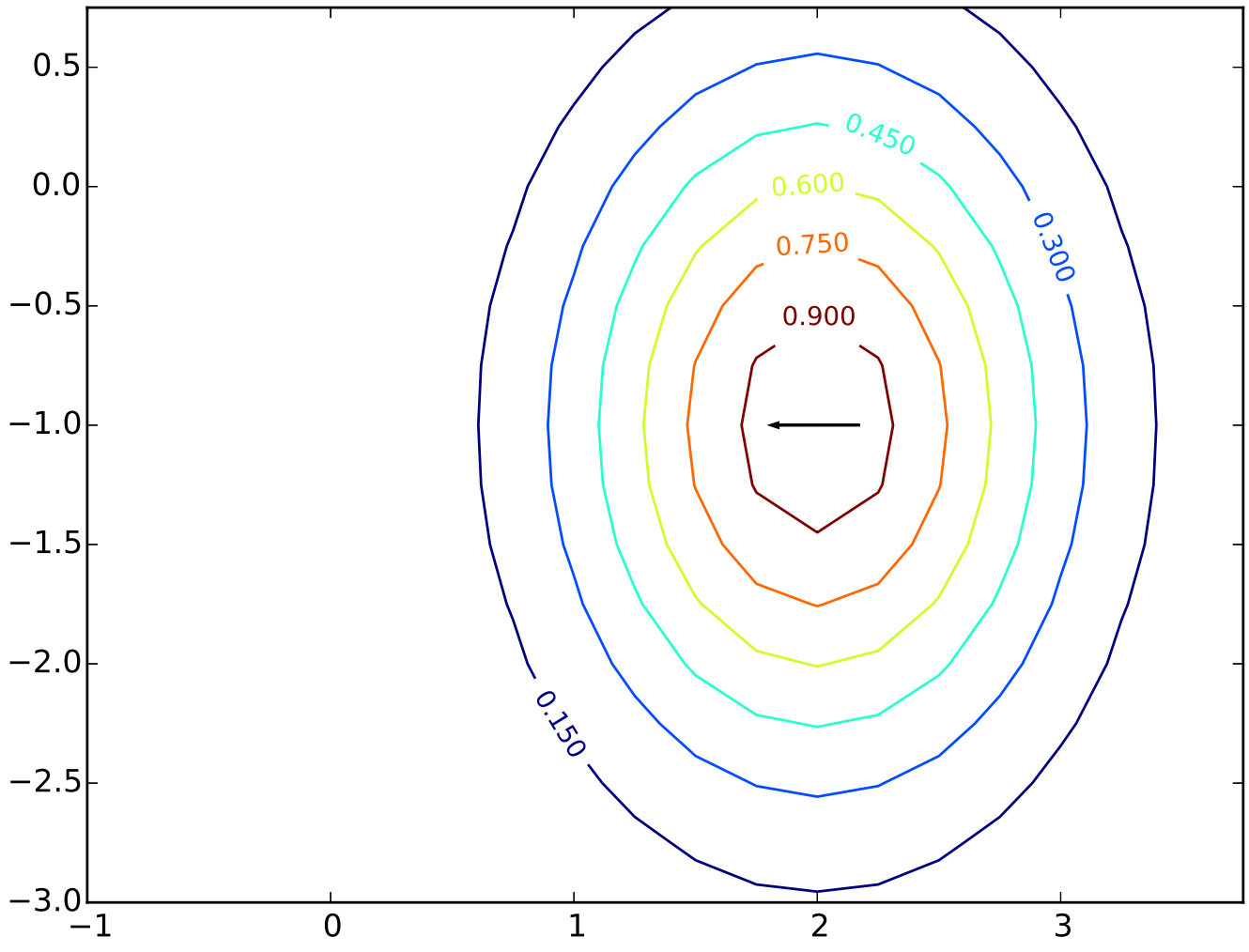
gradient ascent example



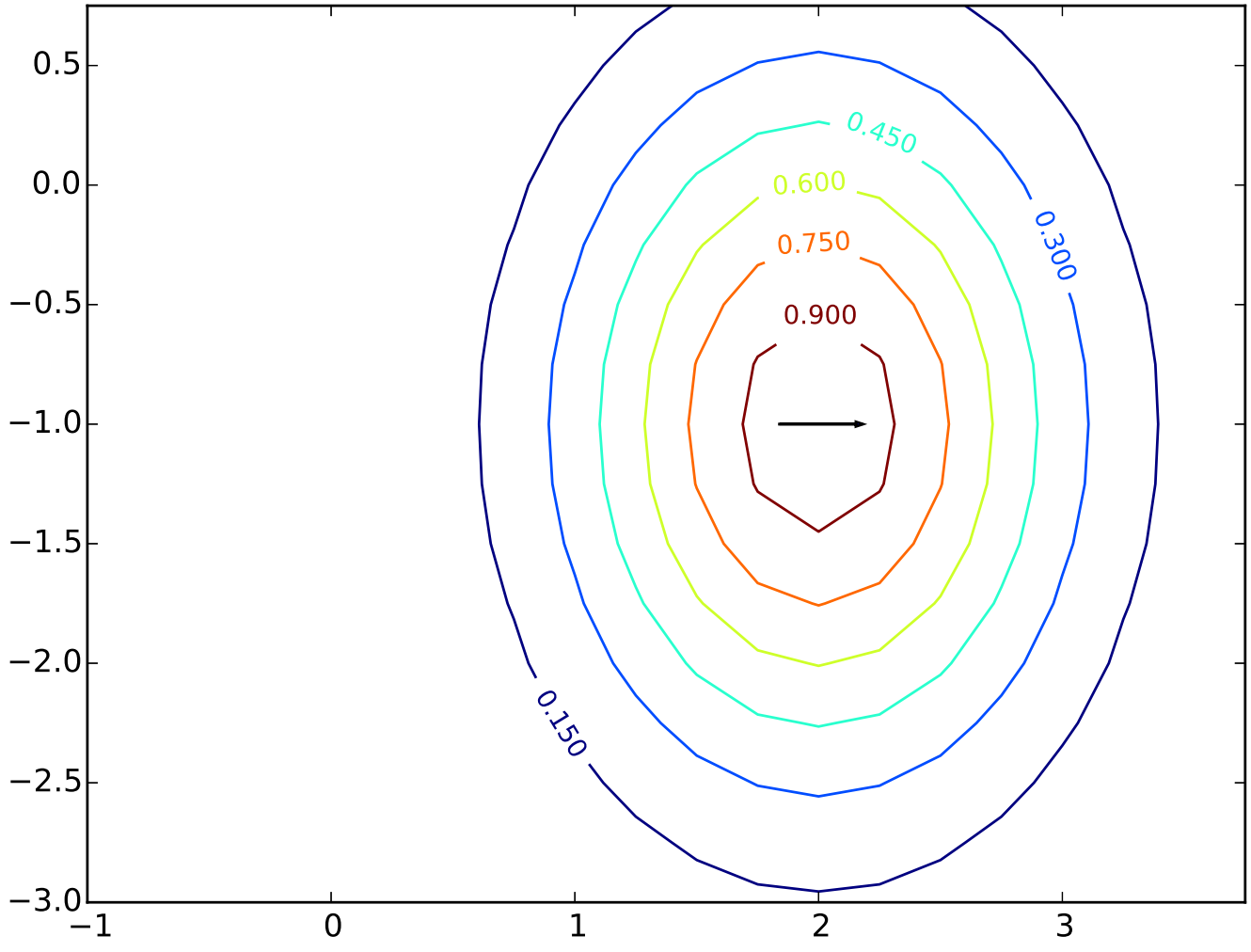
gradient ascent example



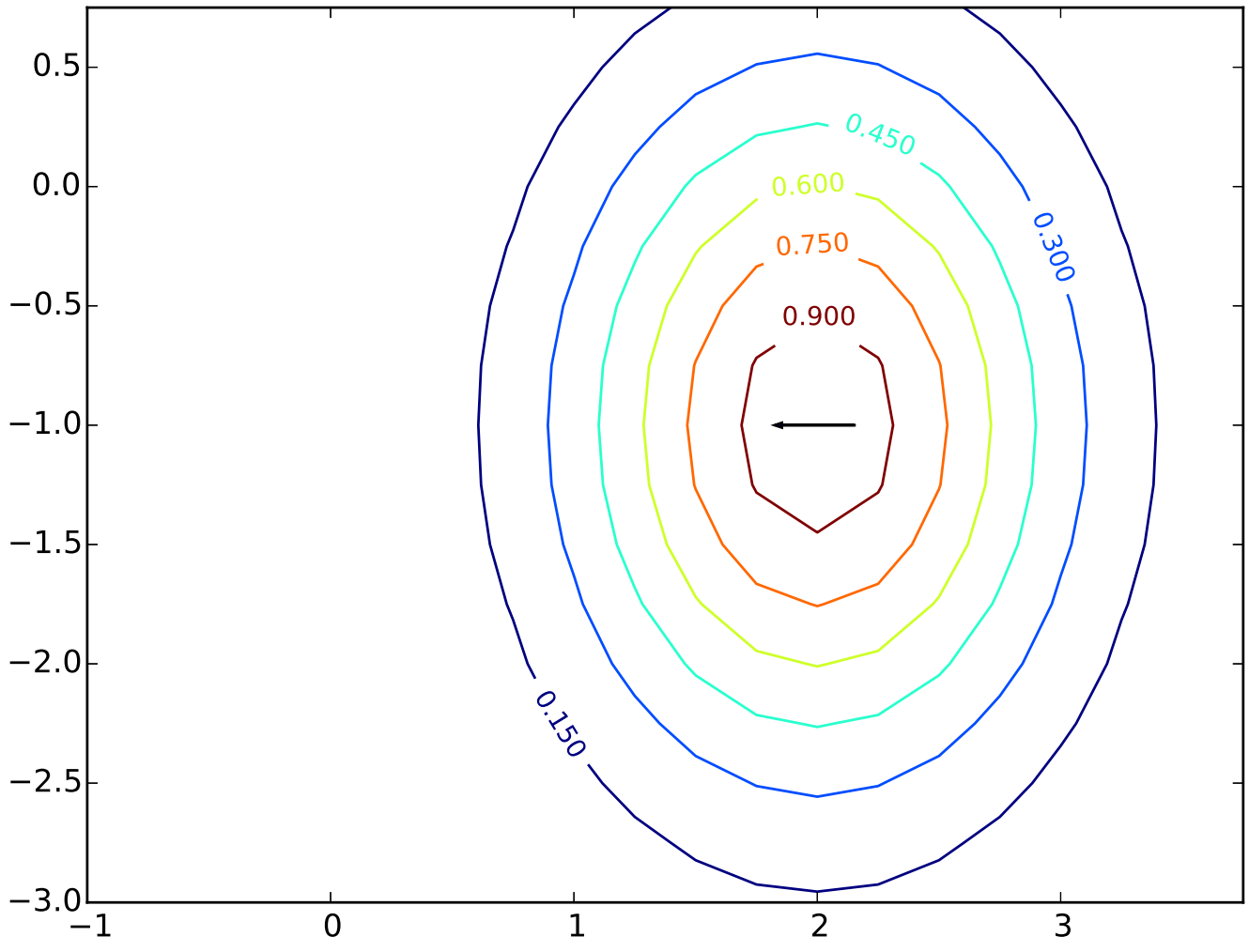
gradient ascent example



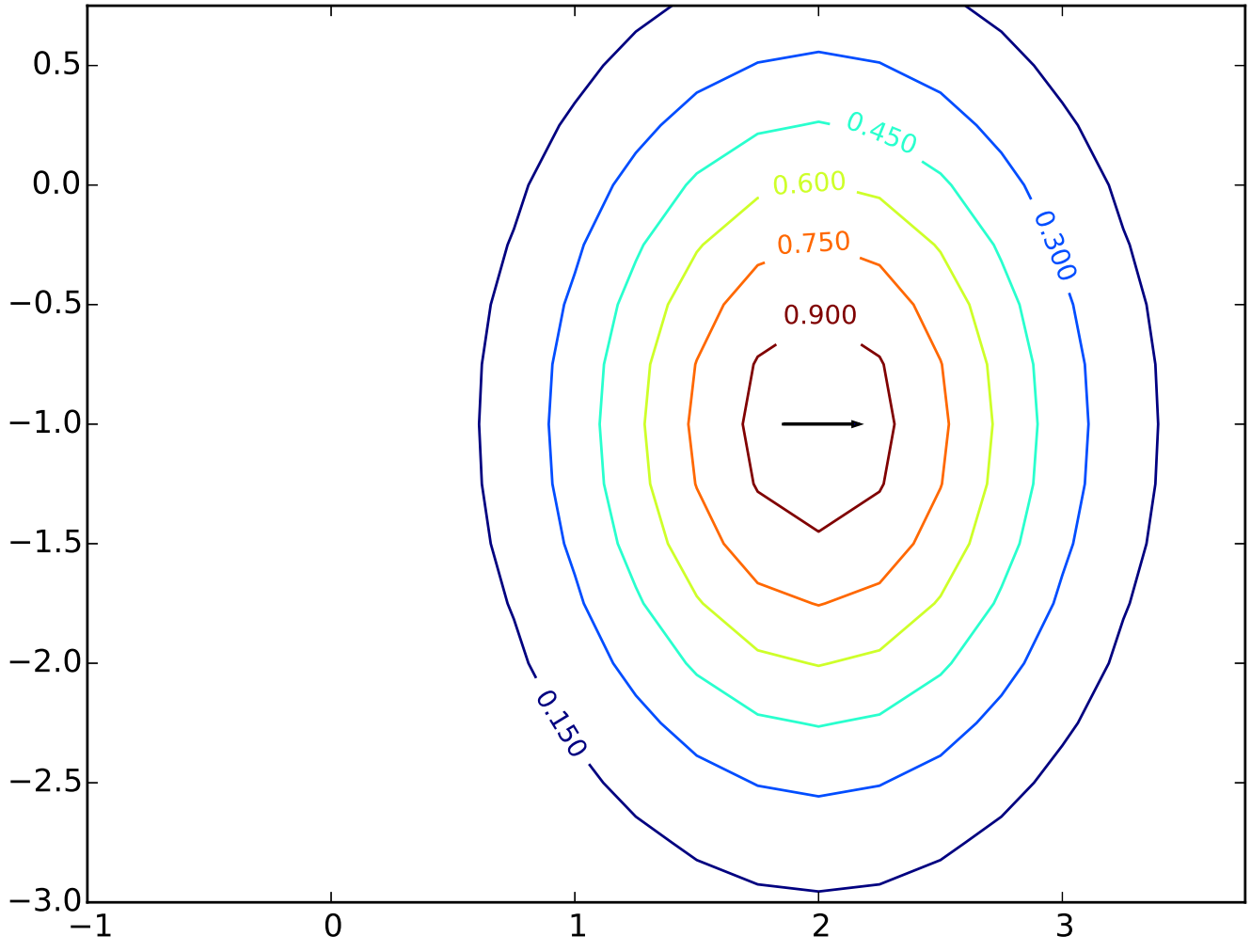
gradient ascent example



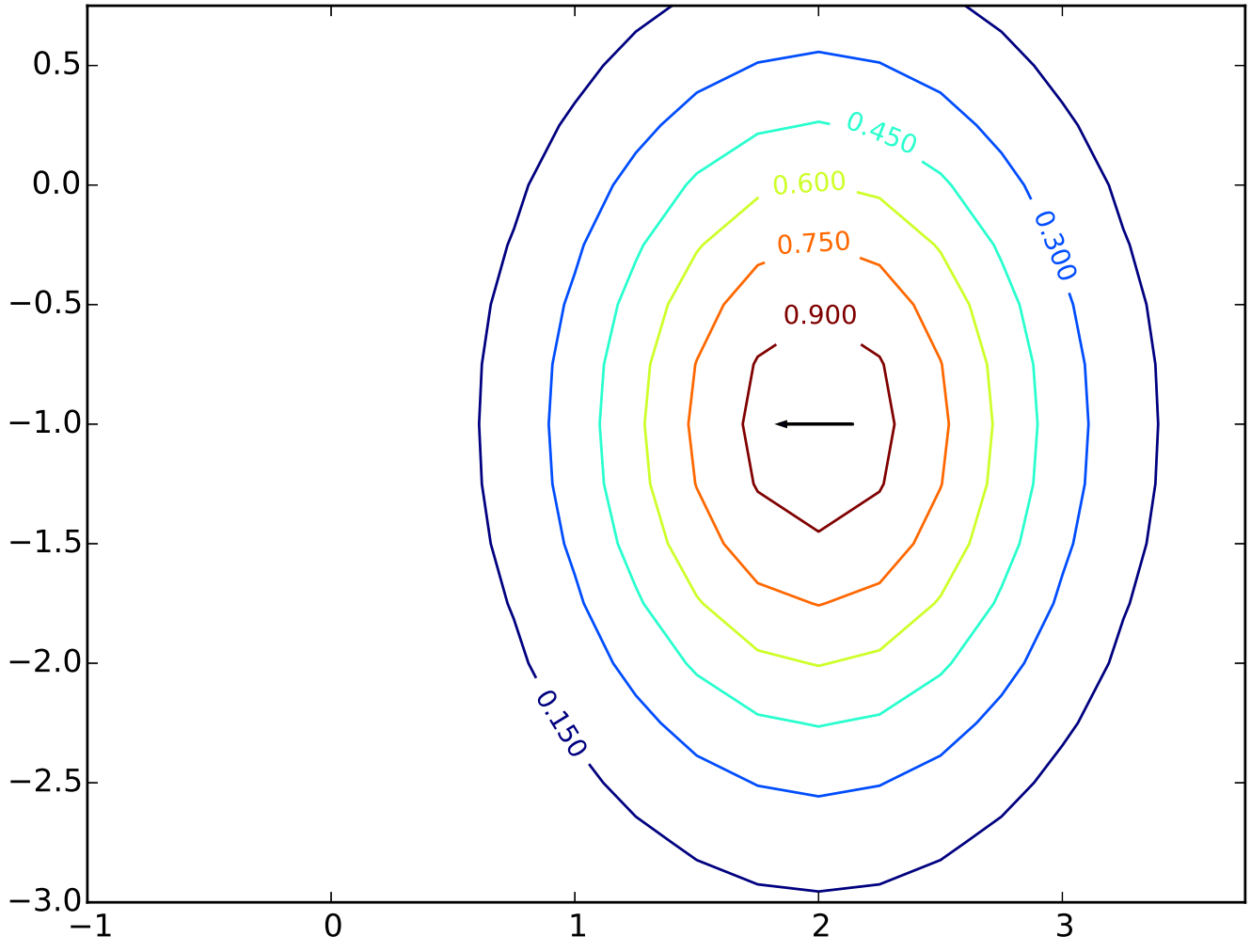
gradient ascent example



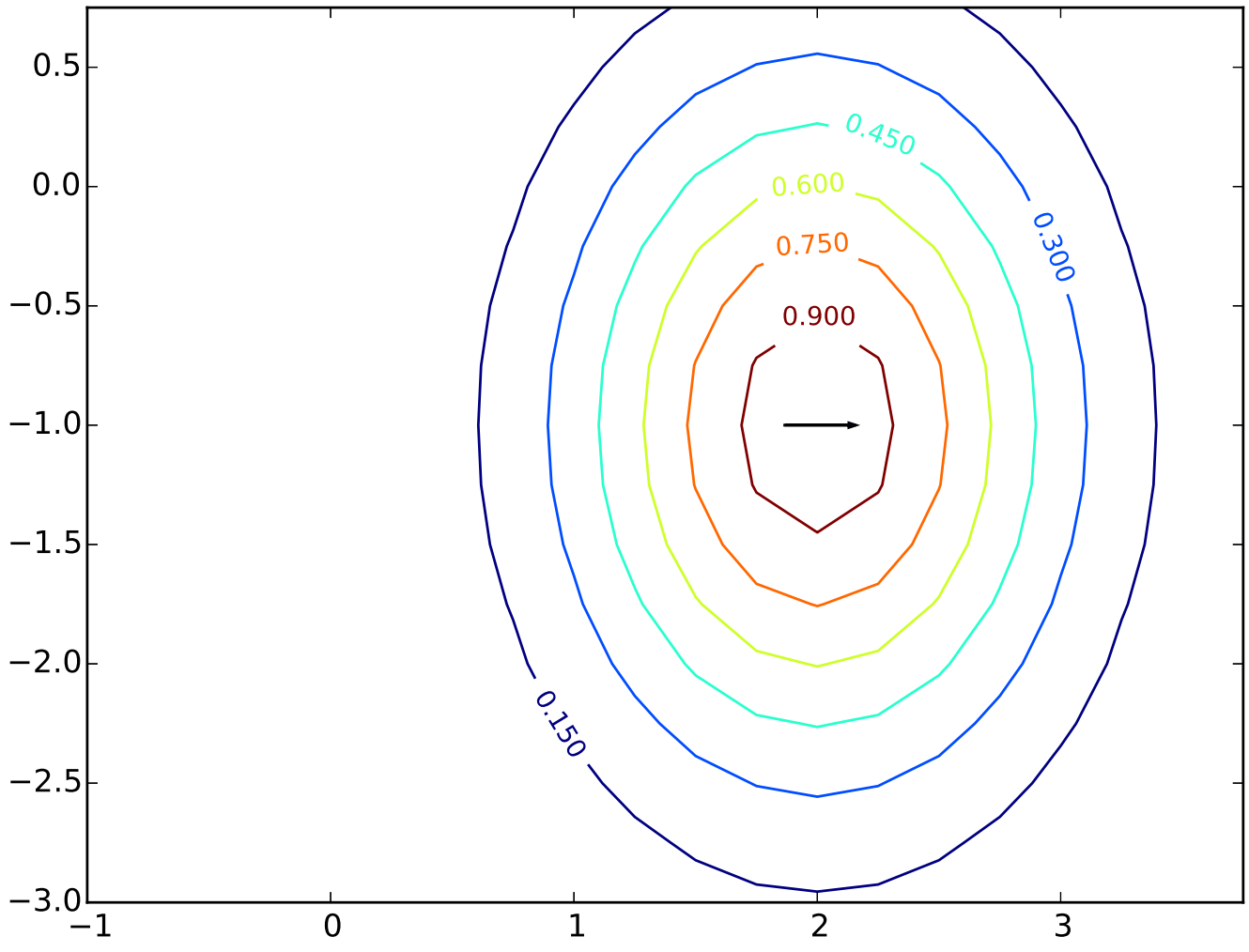
gradient ascent example



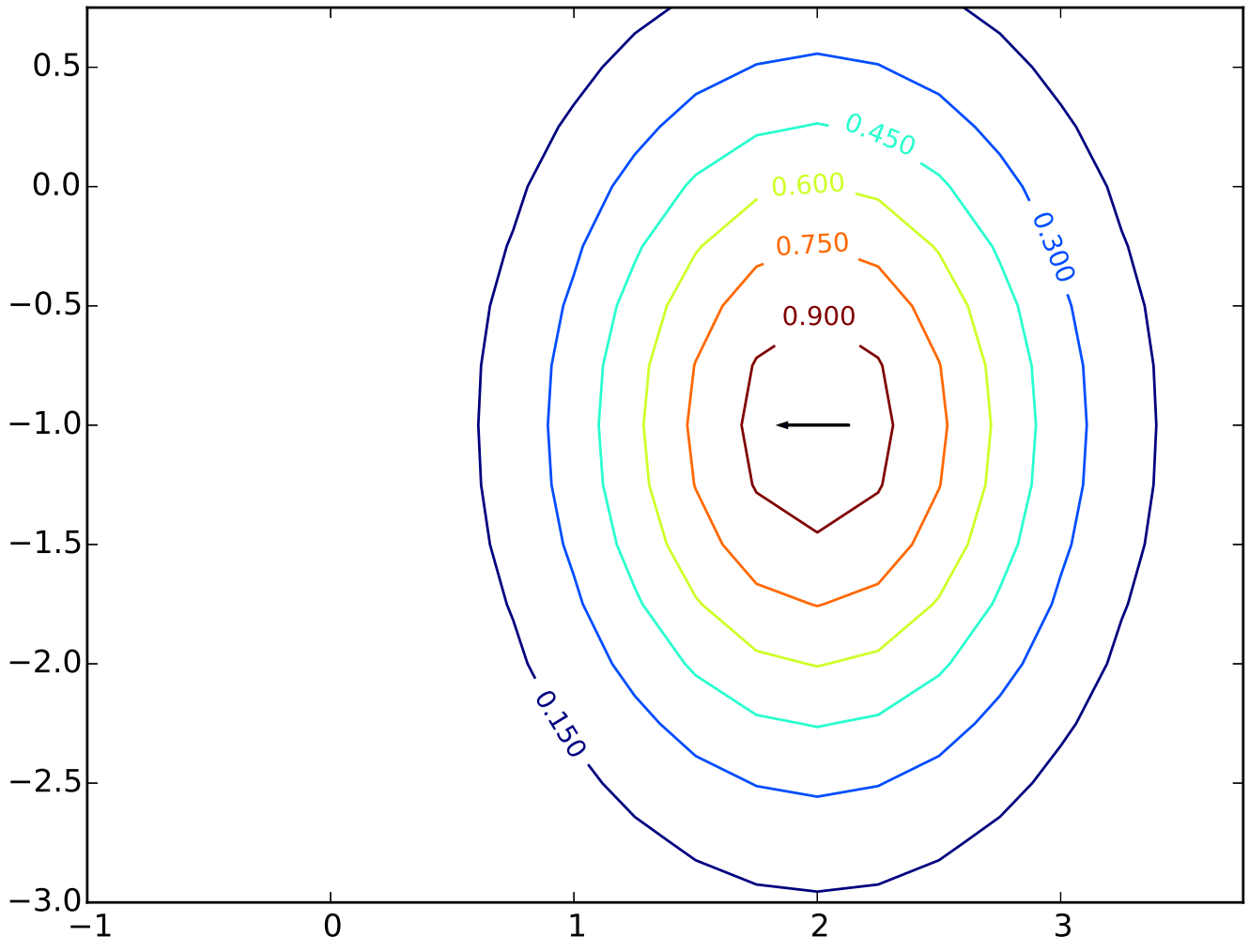
gradient ascent example



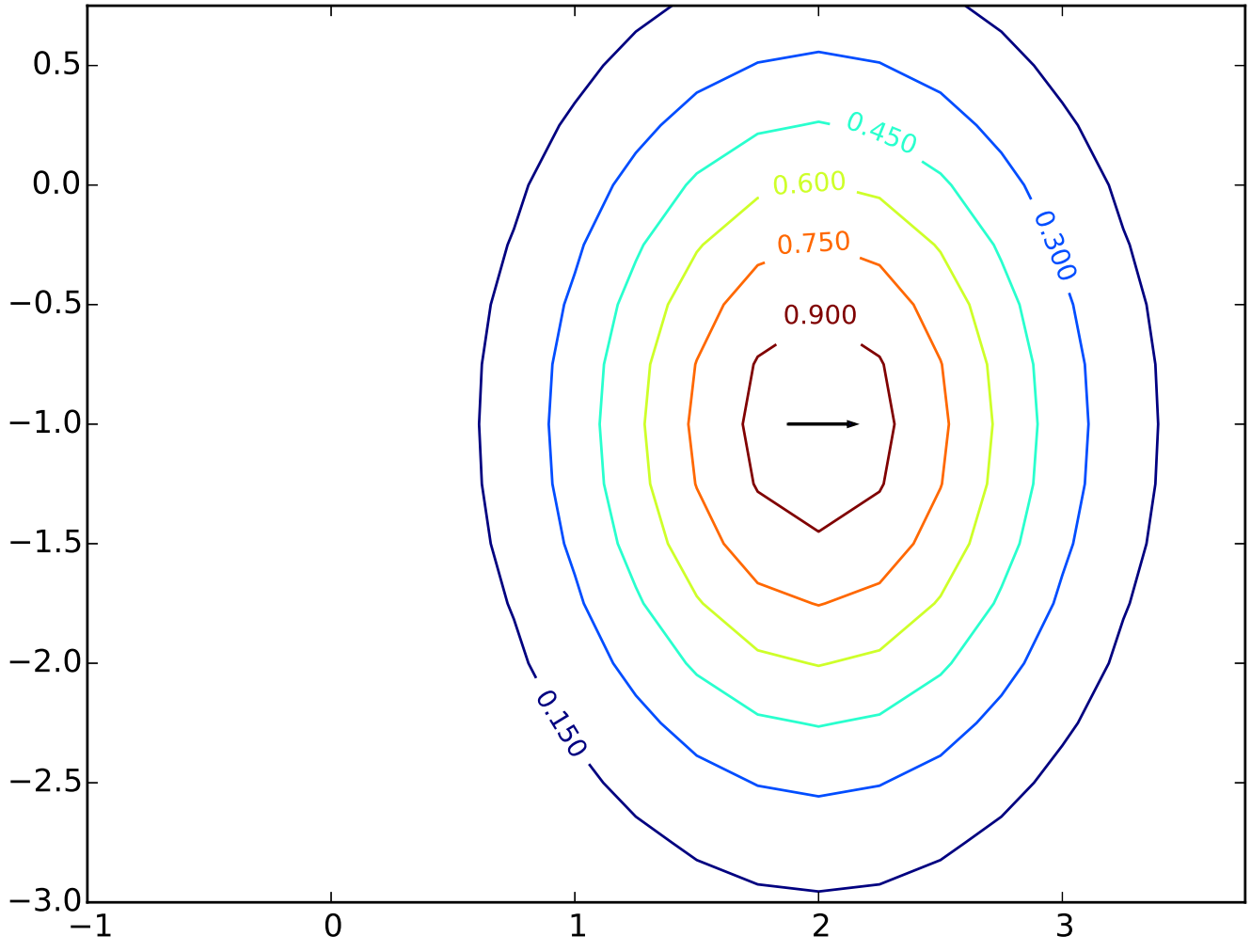
gradient ascent example



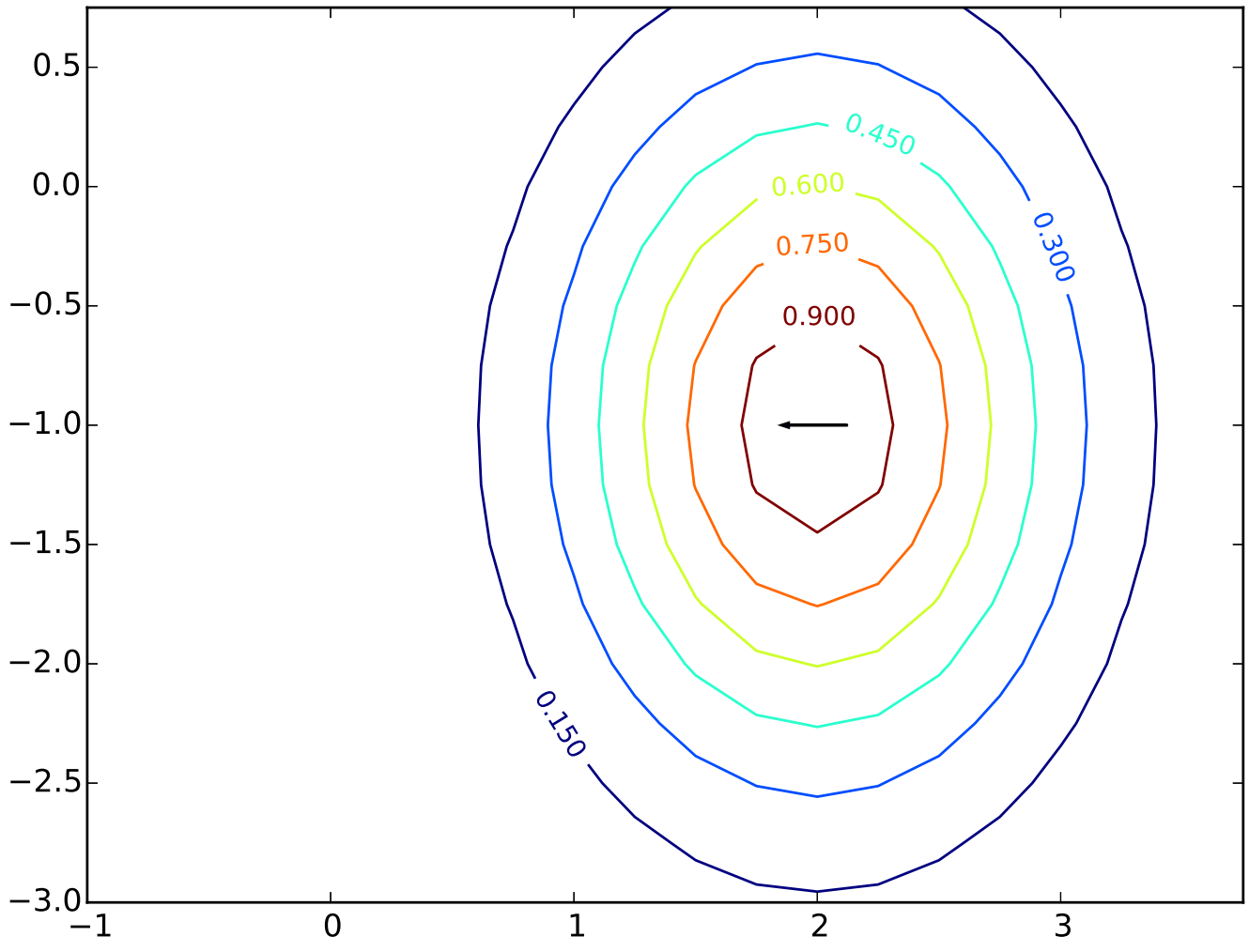
gradient ascent example



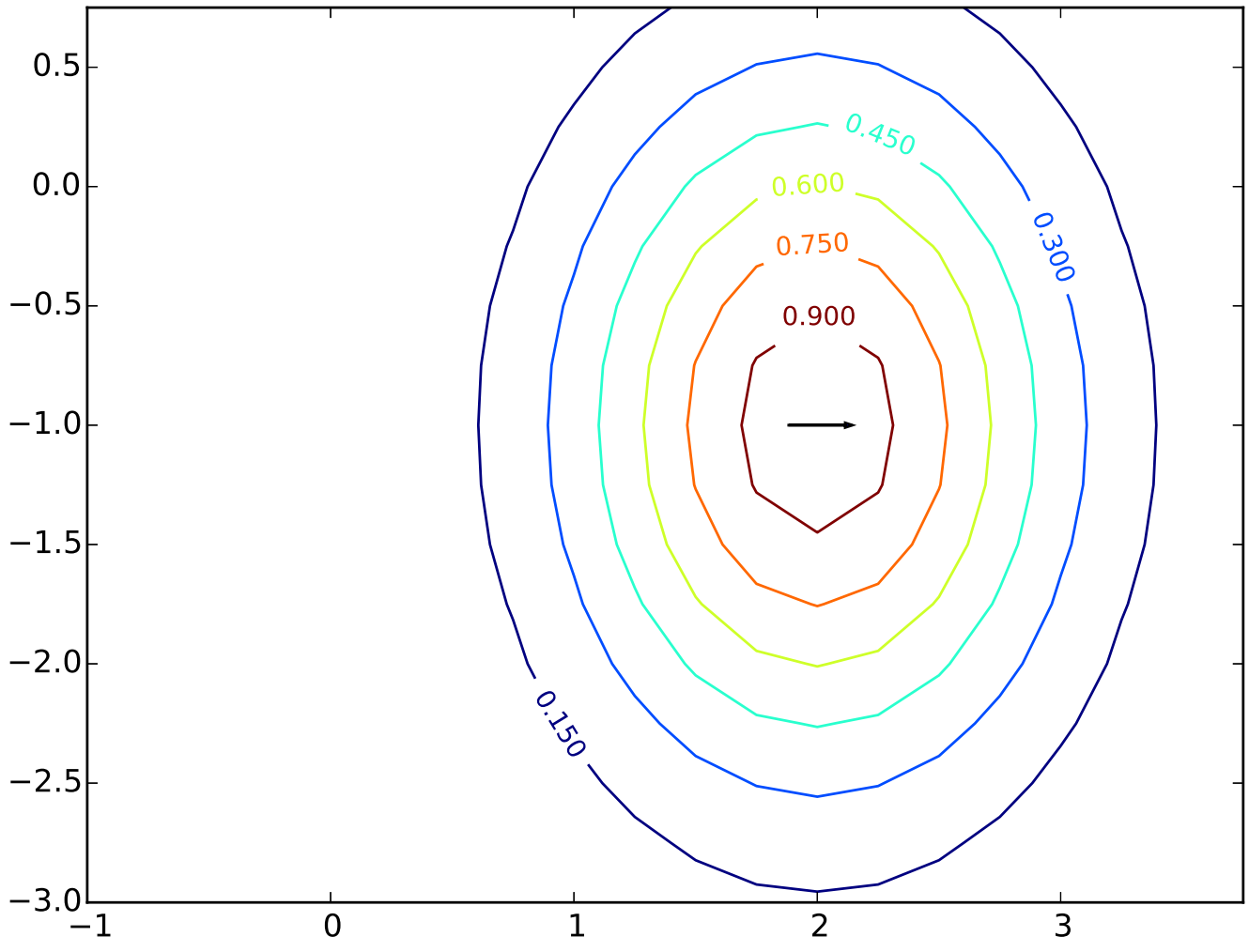
gradient ascent example



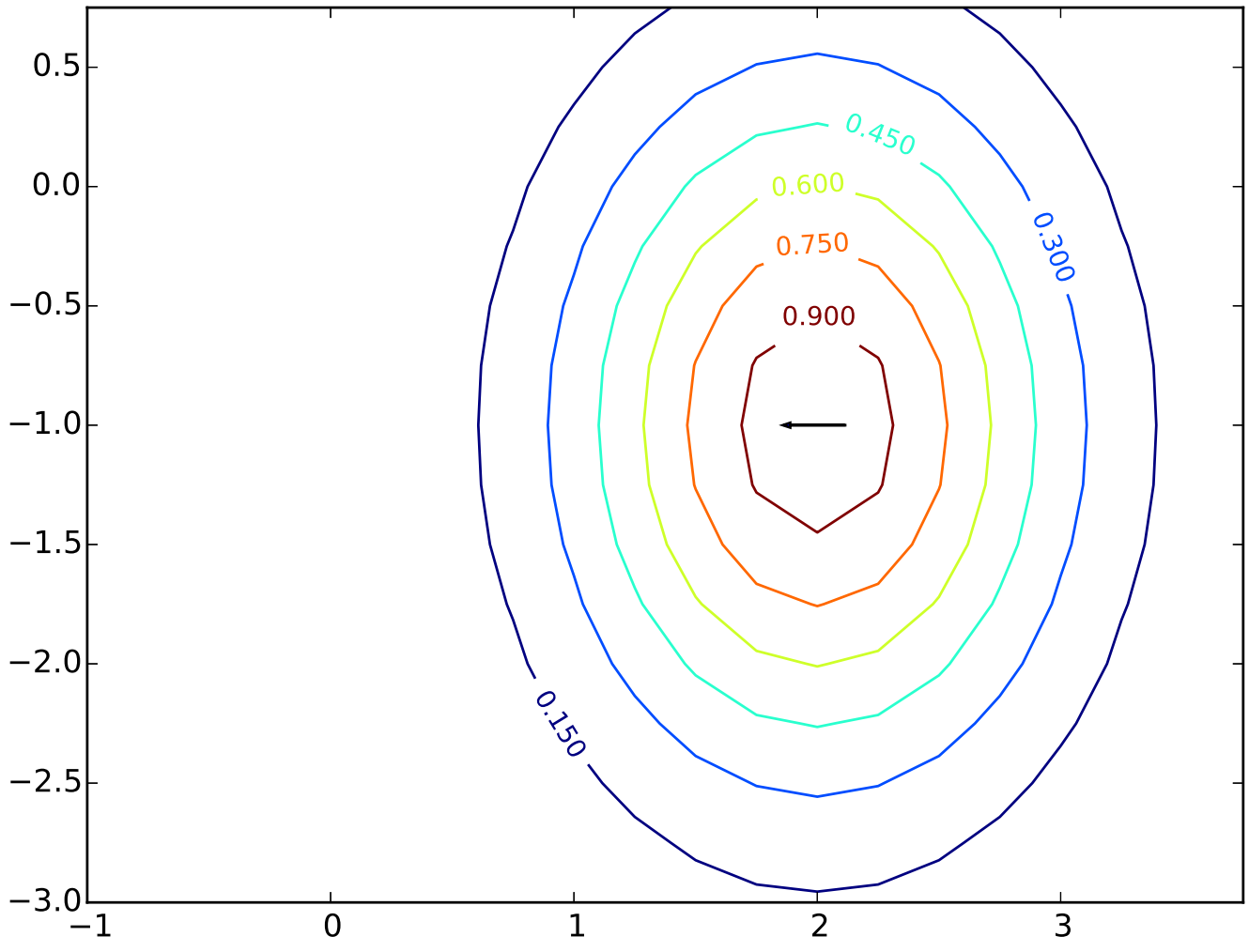
gradient ascent example



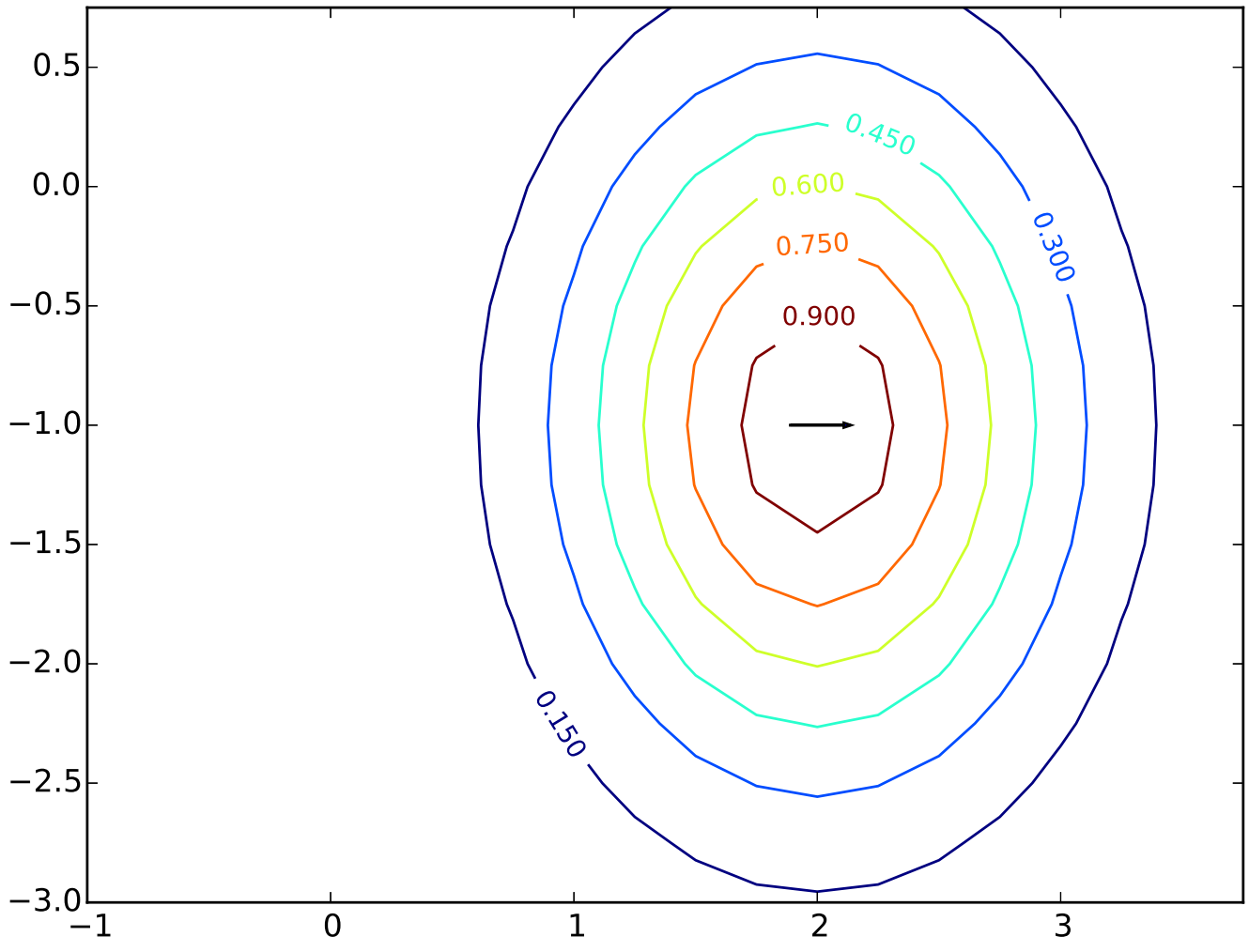
gradient ascent example



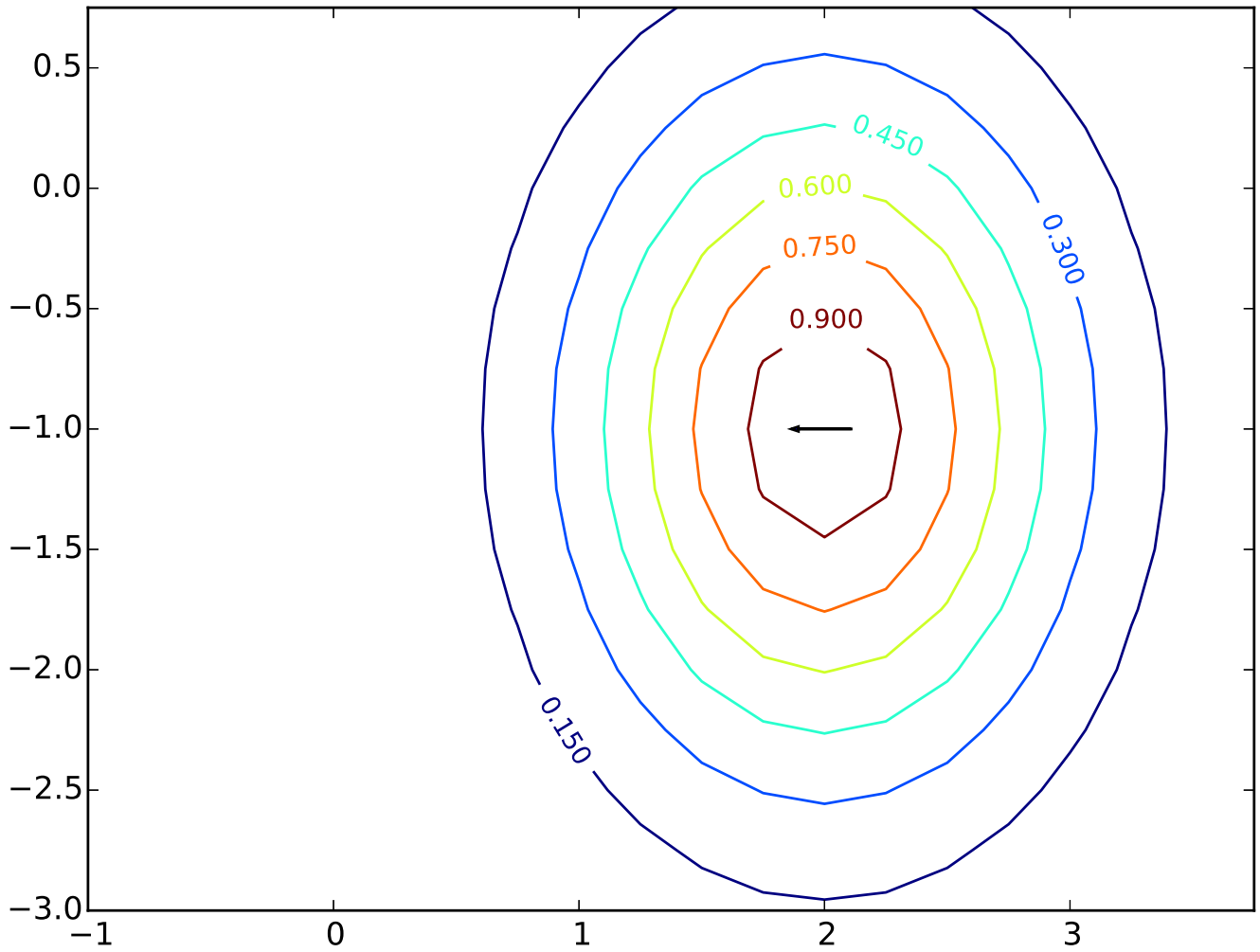
gradient ascent example



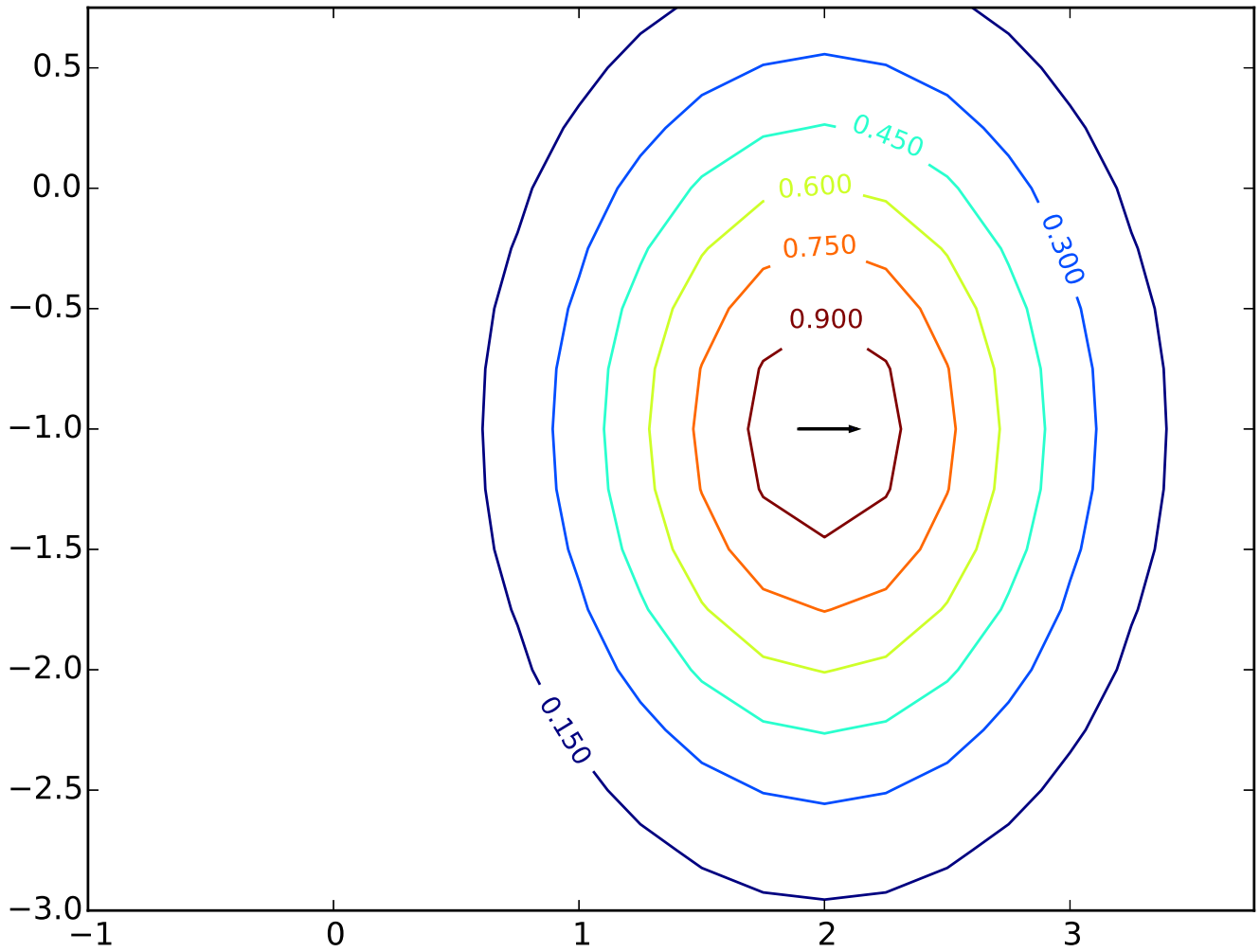
gradient ascent example



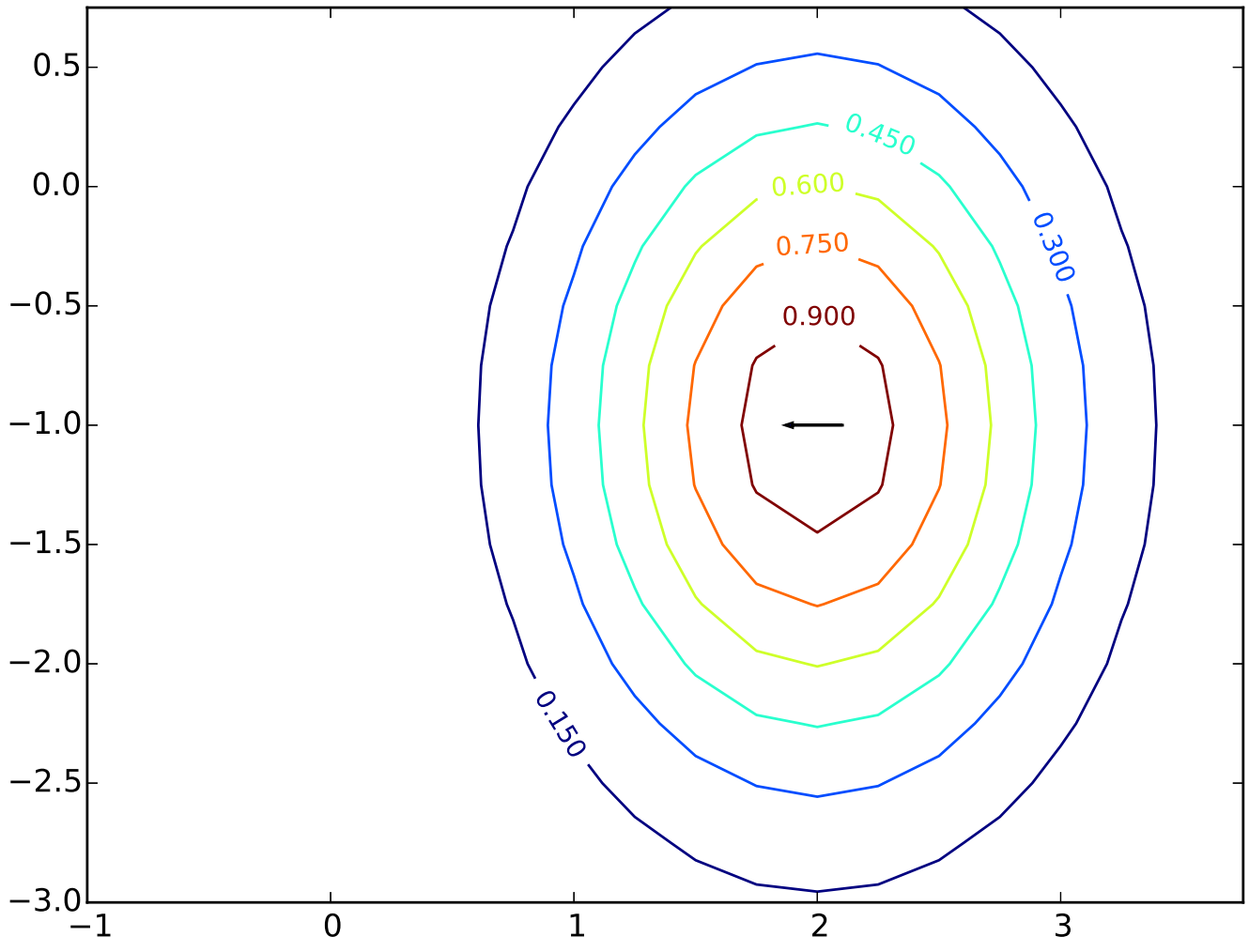
gradient ascent example



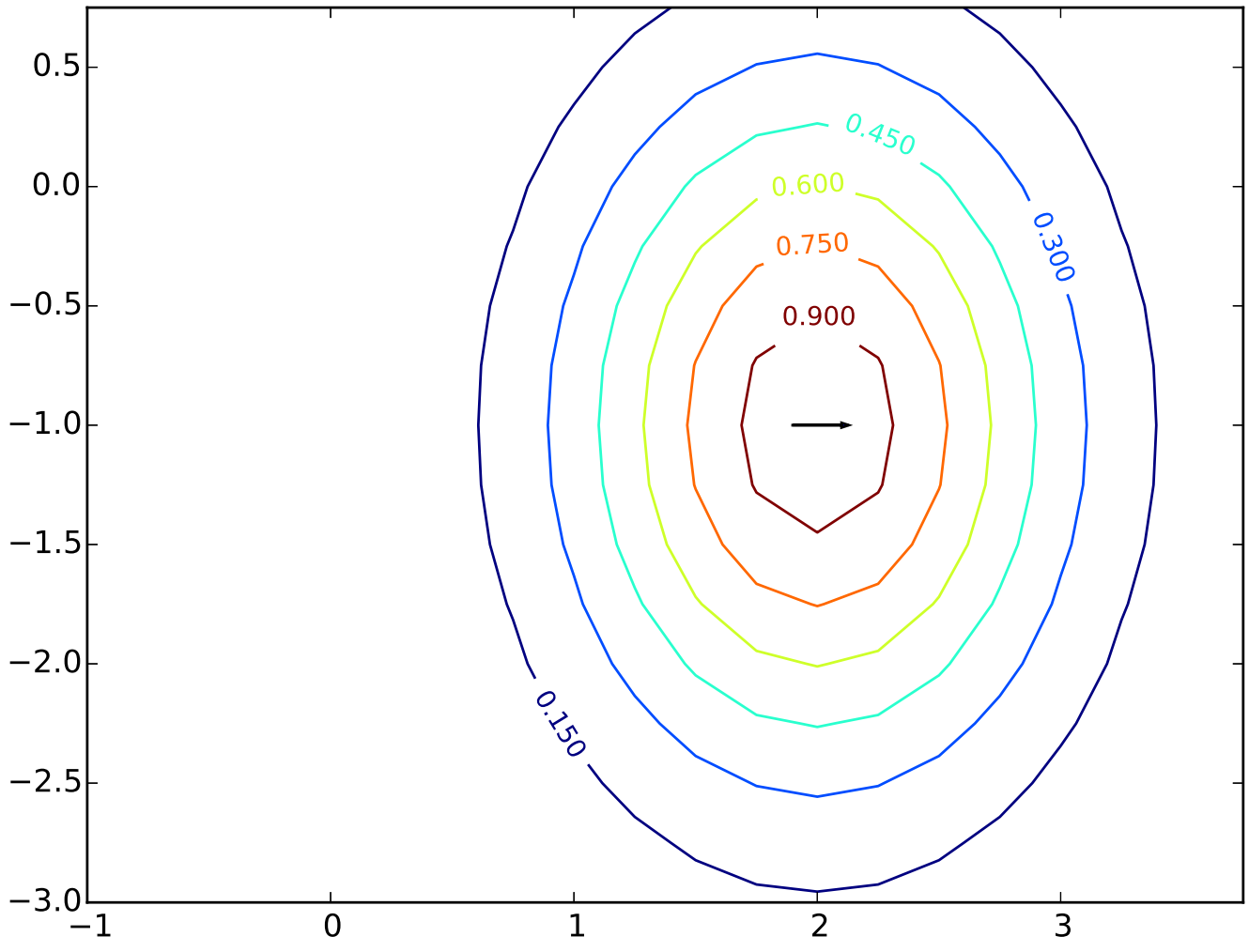
gradient ascent example



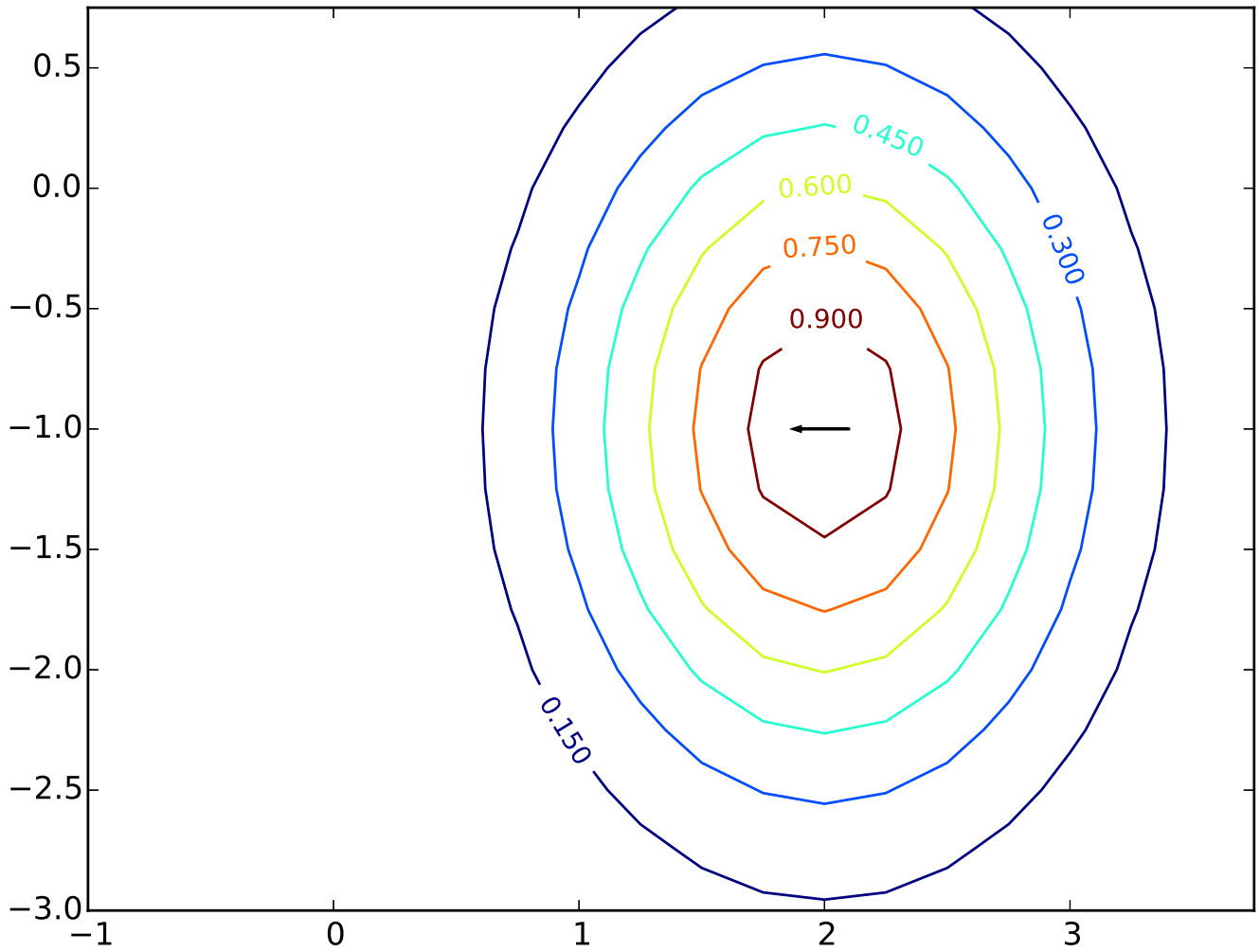
gradient ascent example



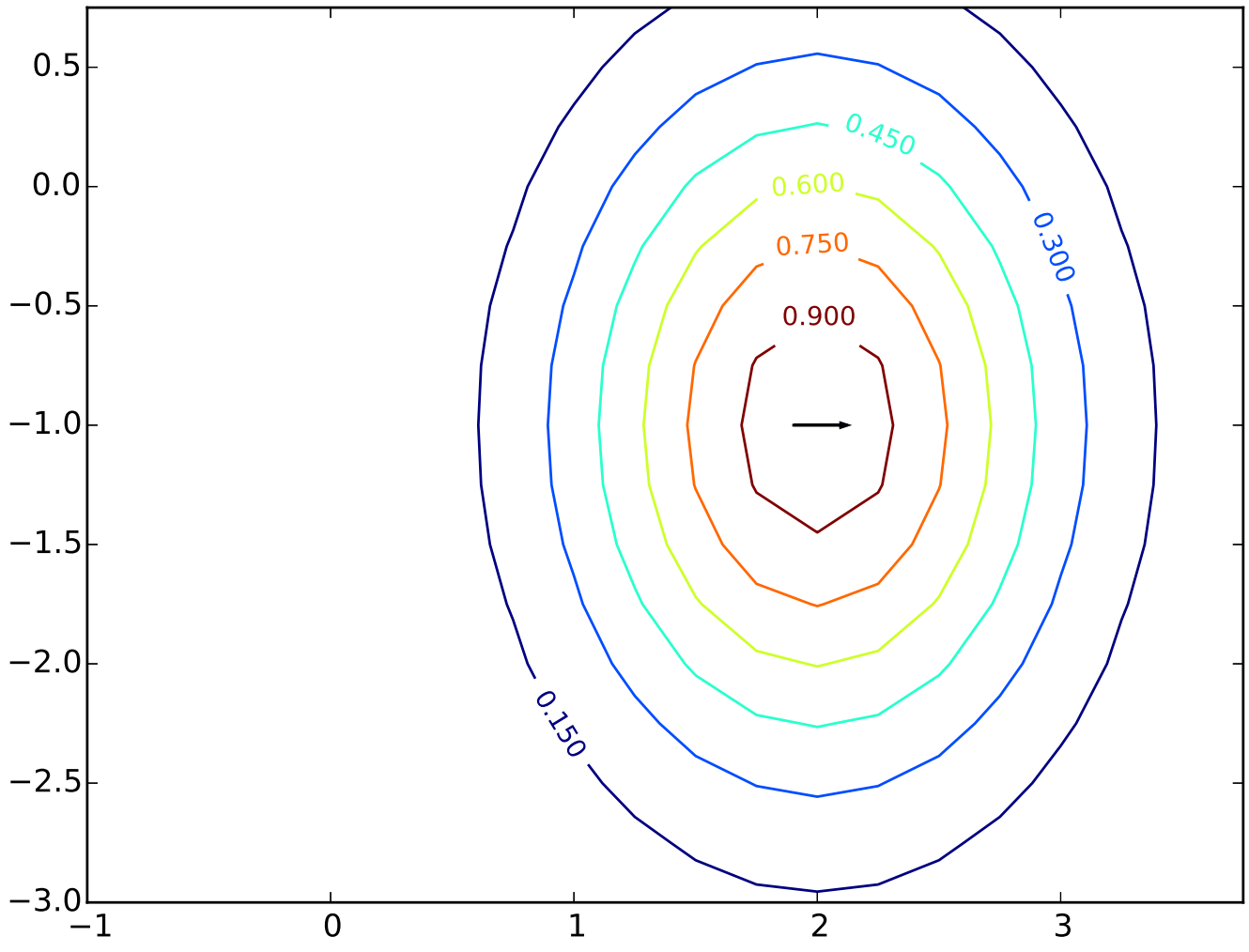
gradient ascent example



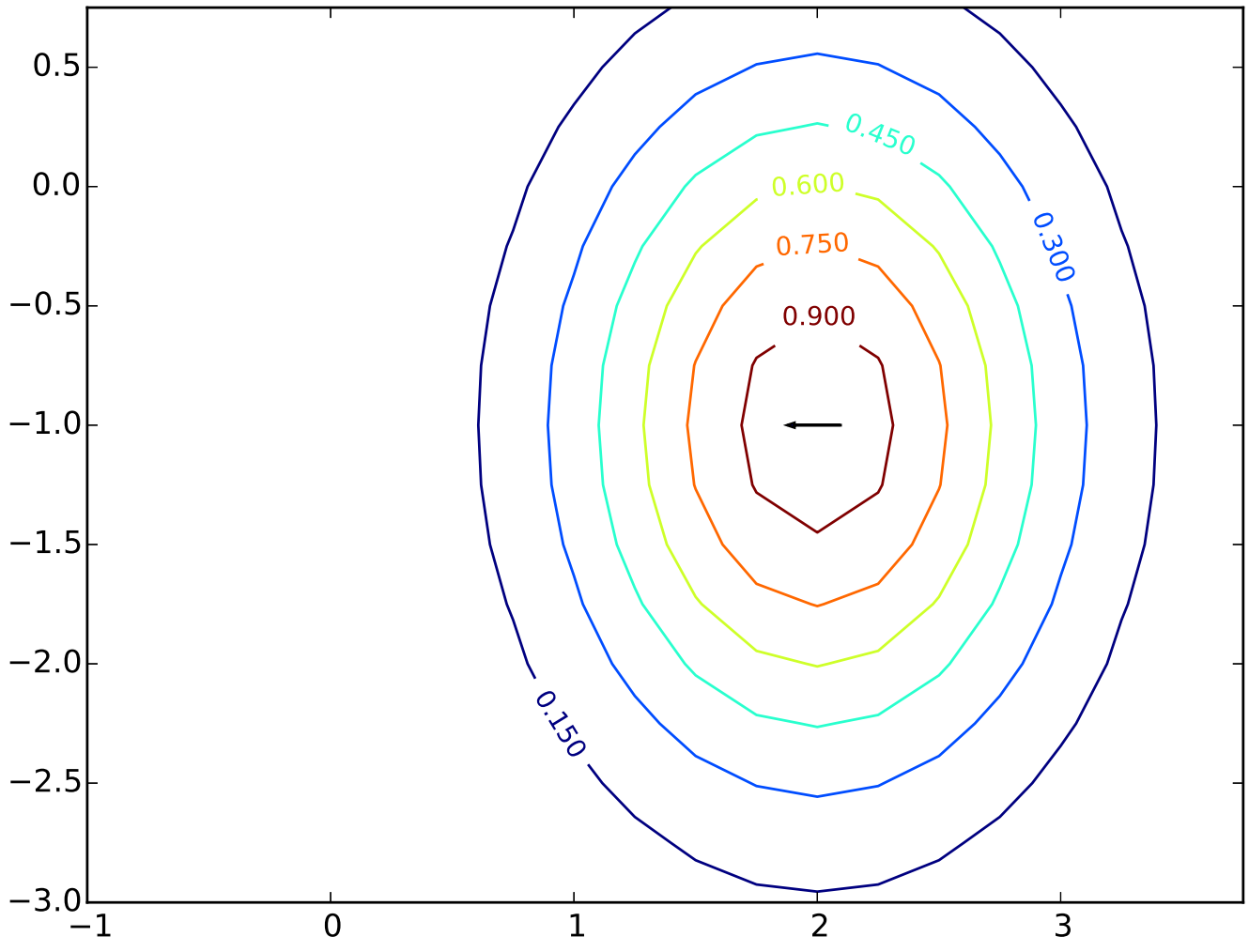
gradient ascent example



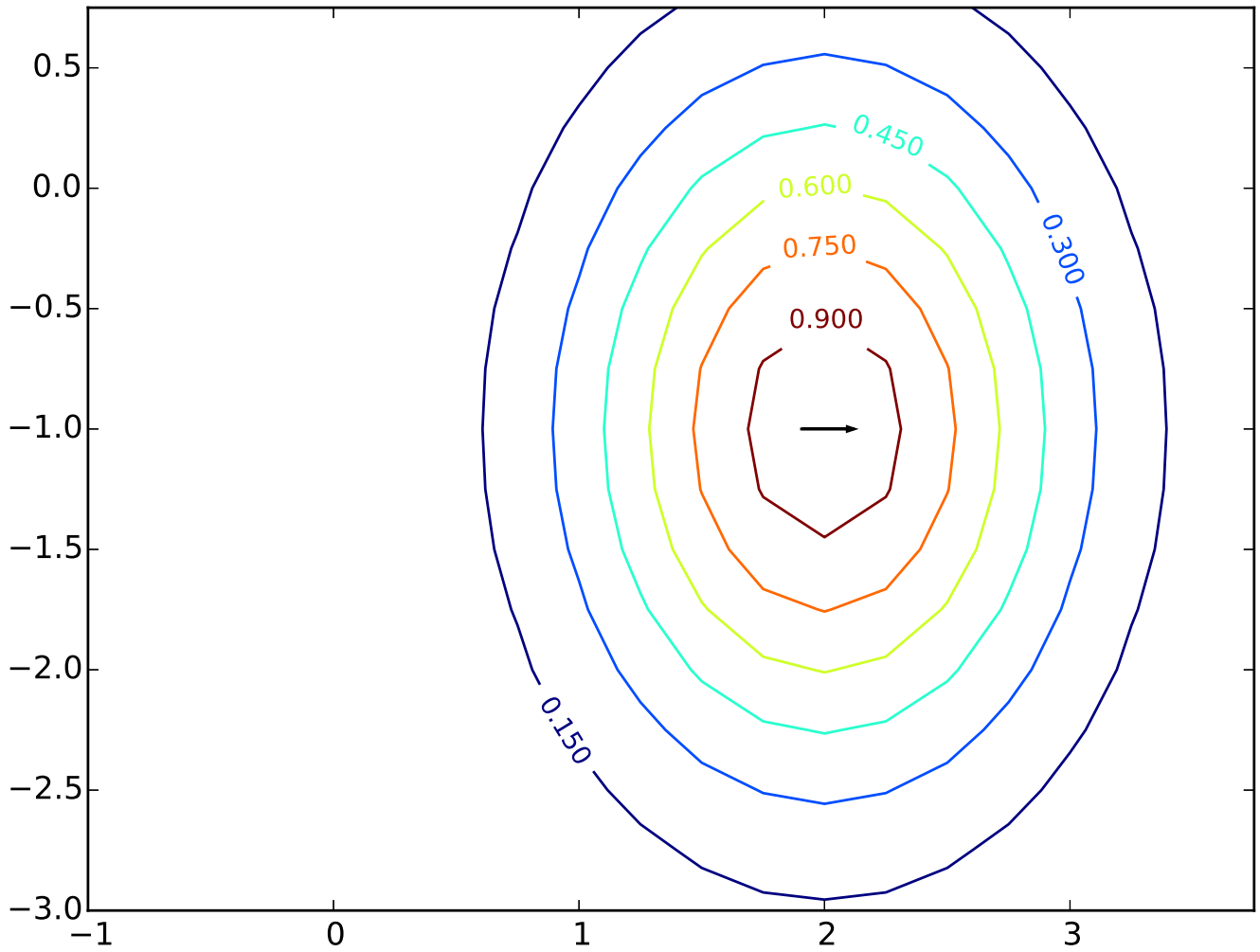
gradient ascent example



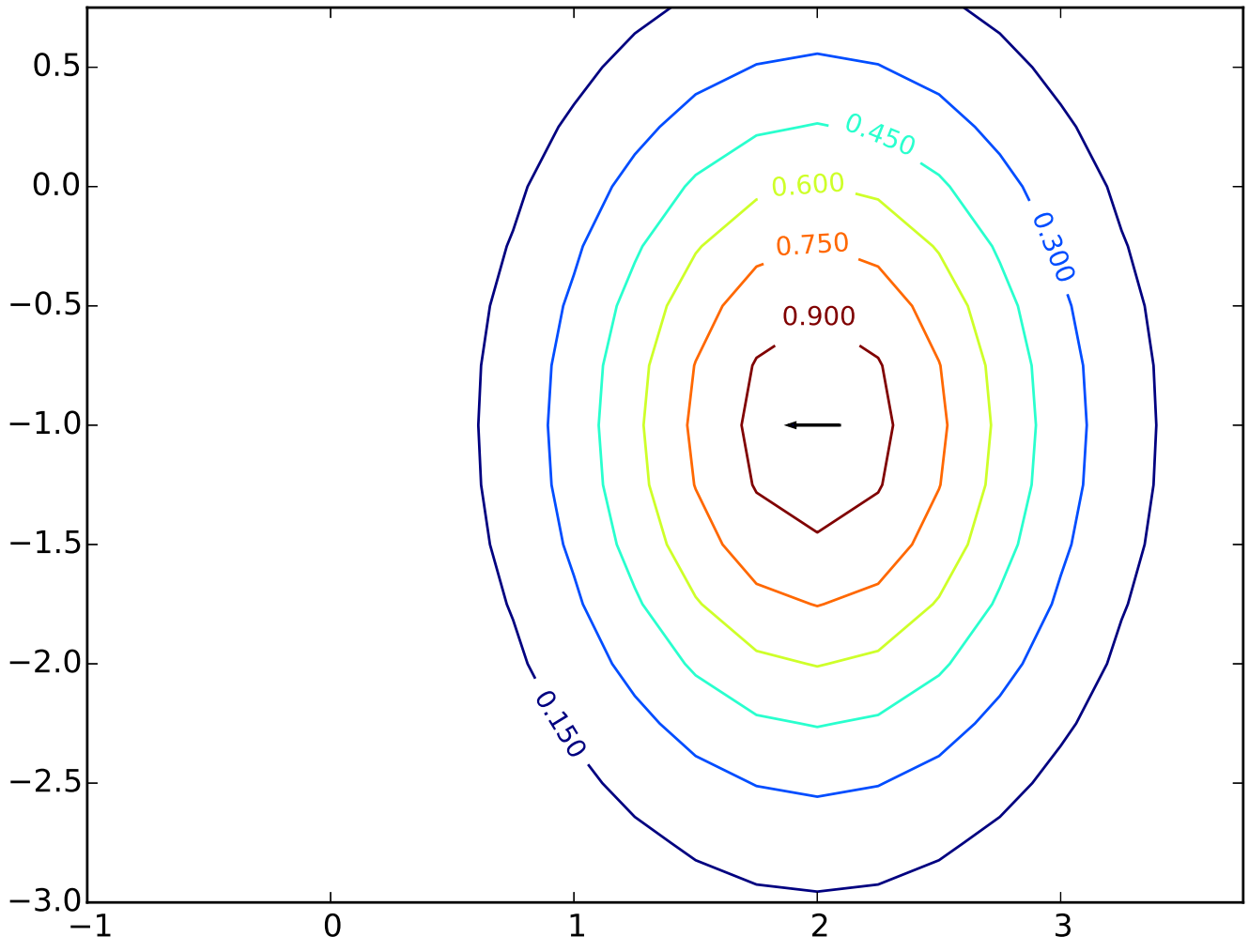
gradient ascent example



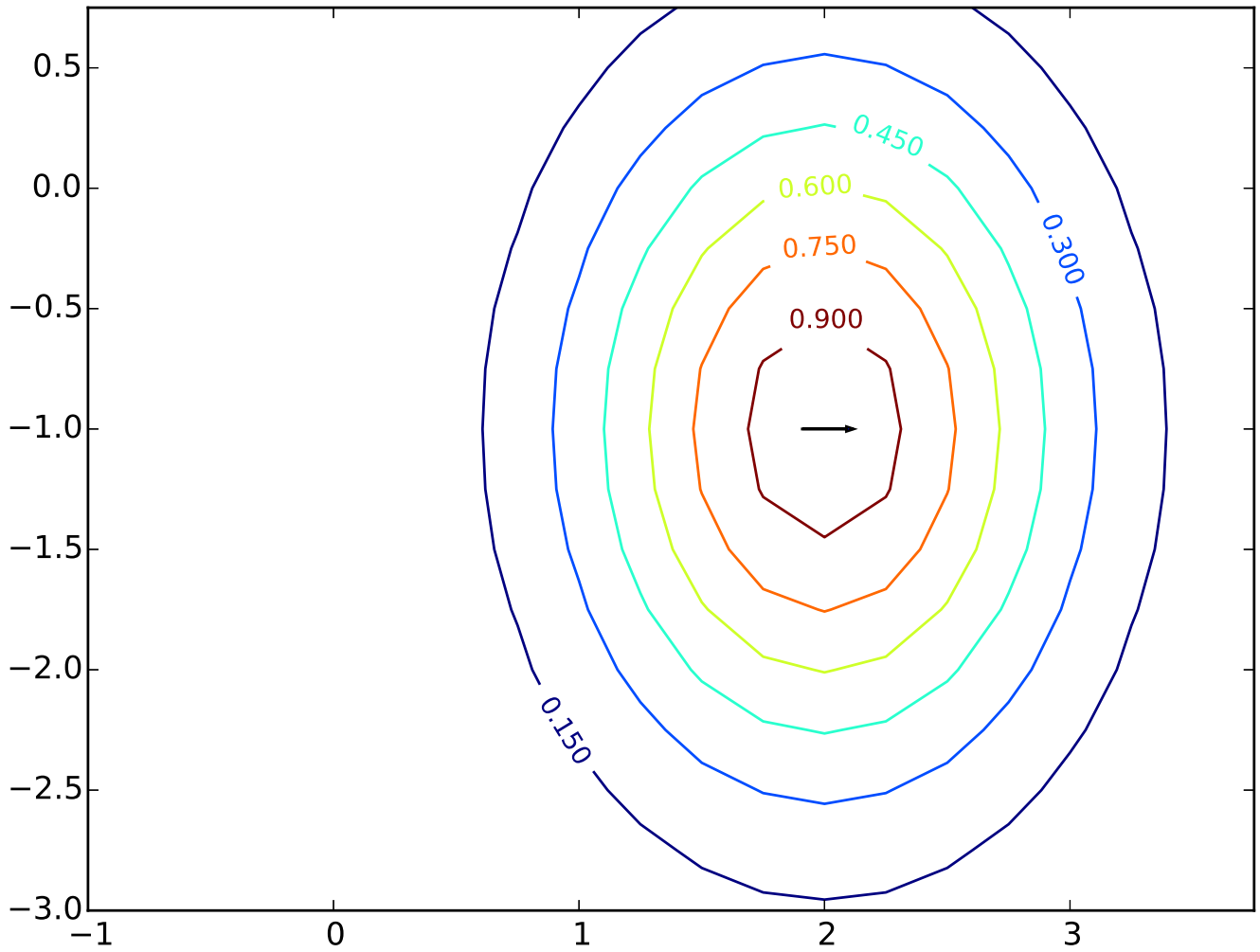
gradient ascent example



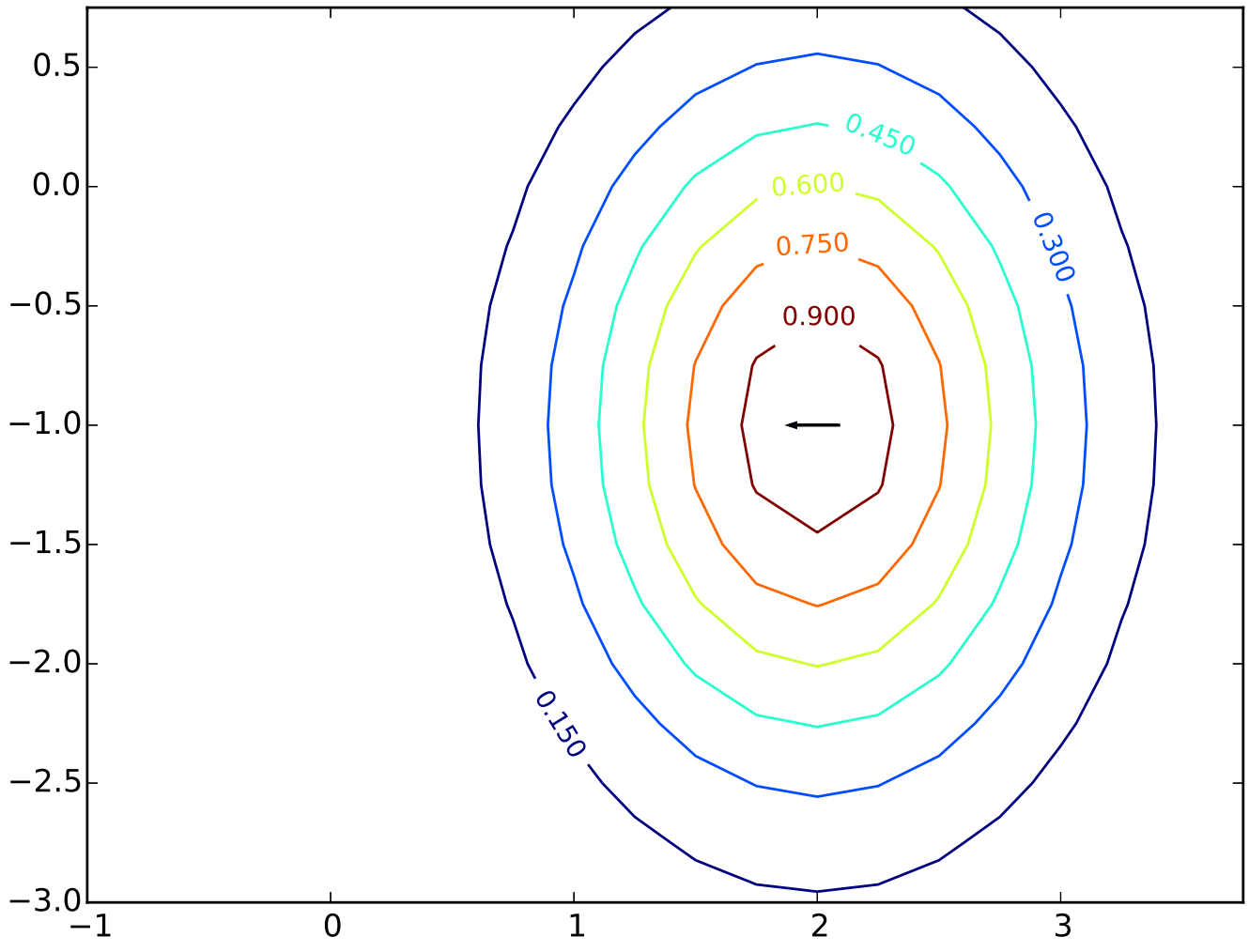
gradient ascent example



gradient ascent example



gradient ascent example



will we always reach the top?

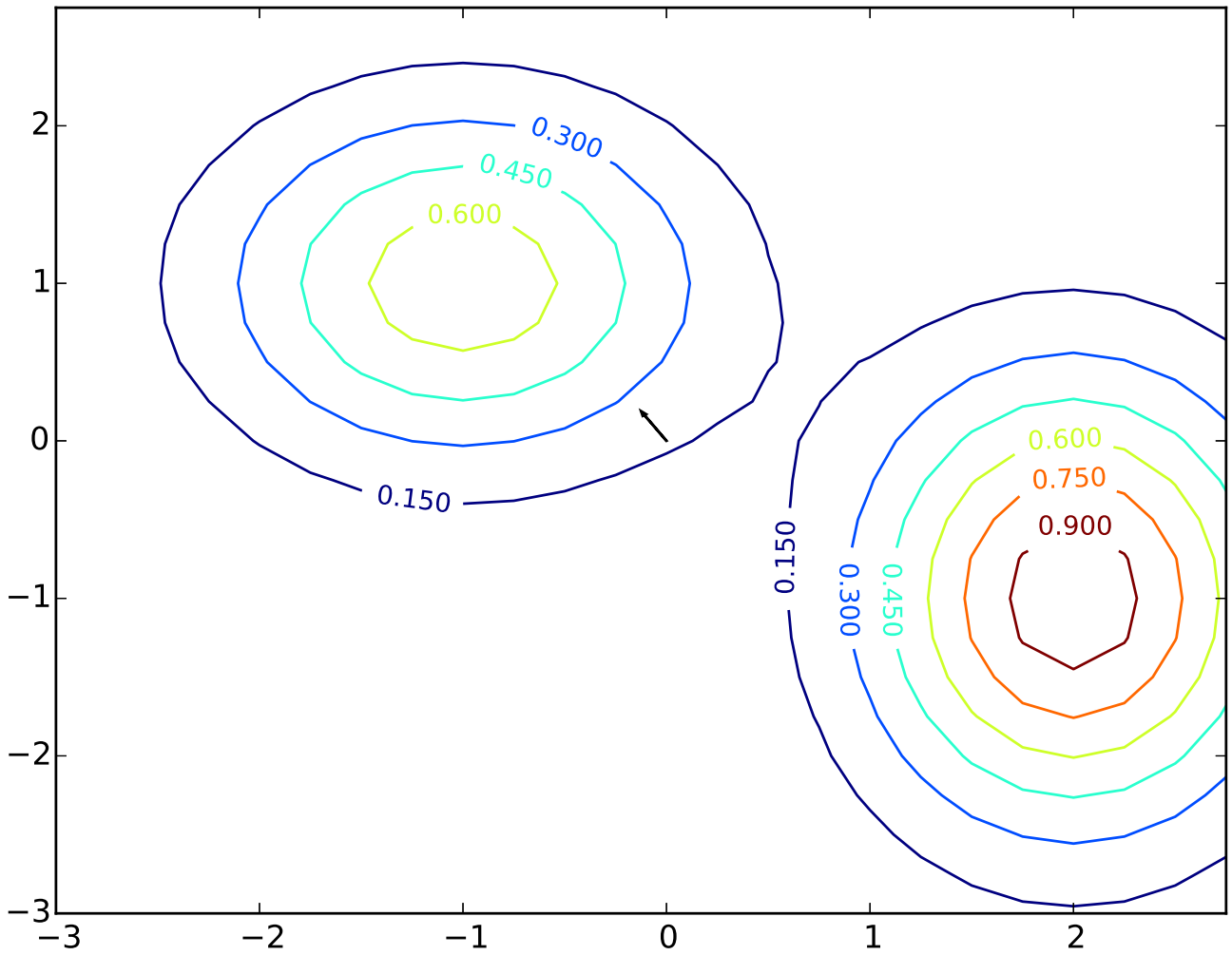
- yes, if
  - there is actually a top
  - the step is short enough
  - the surface isn't too jumpy
- smarter versions of gradient ascent/descent try to adapt the step length so that we don't go too slow in the beginning, or bounce around the top at the end

gradient ascent example (2)

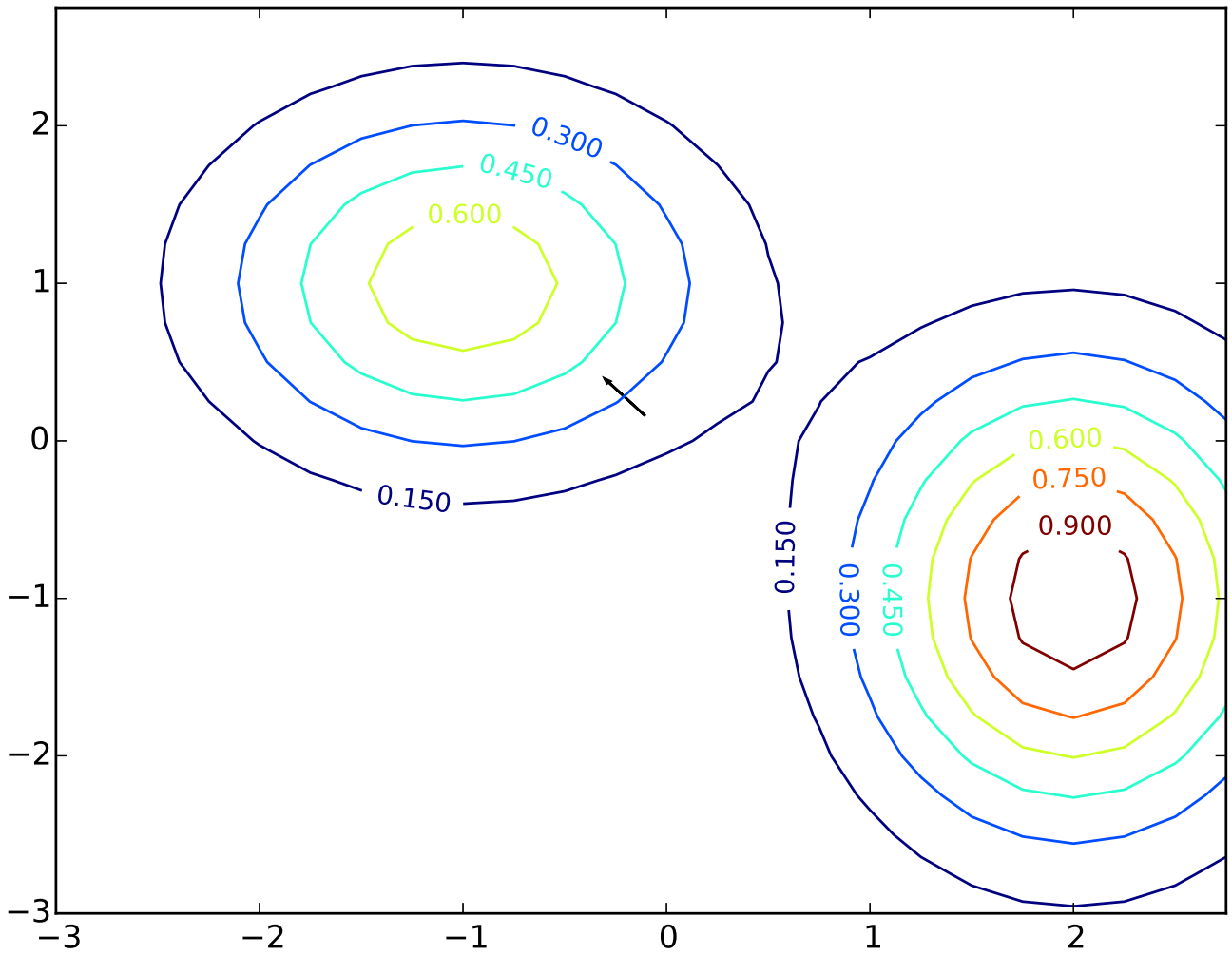
- let's optimize another function:

```
def f(x, y):
    return math.exp( -(x-2)**2 - 0.5*(y+1)**2) \
           + 0.7 * math.exp( -0.7*(x+1)**2 - 0.8*(y-1)**2)
```

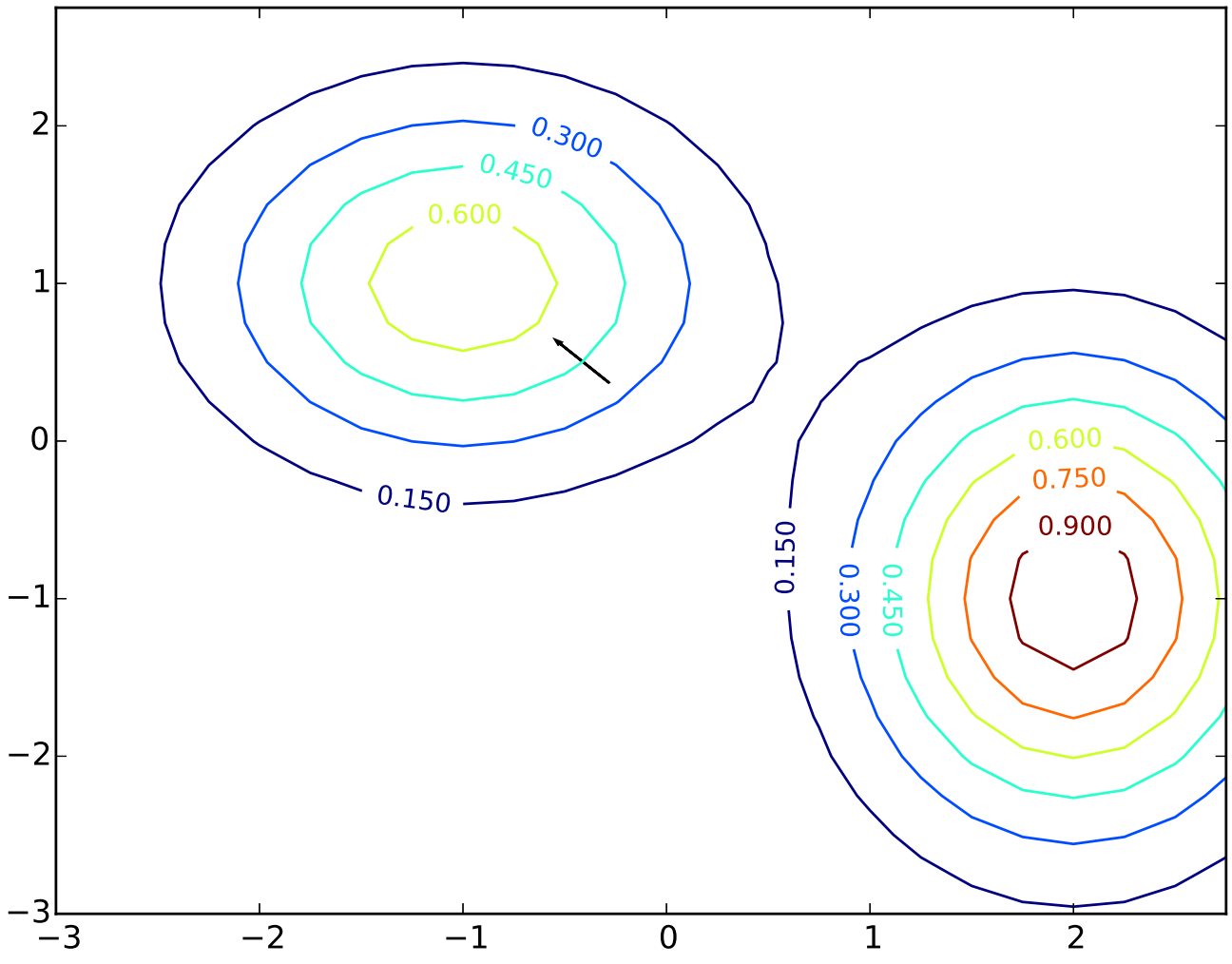
gradient ascent example (2)



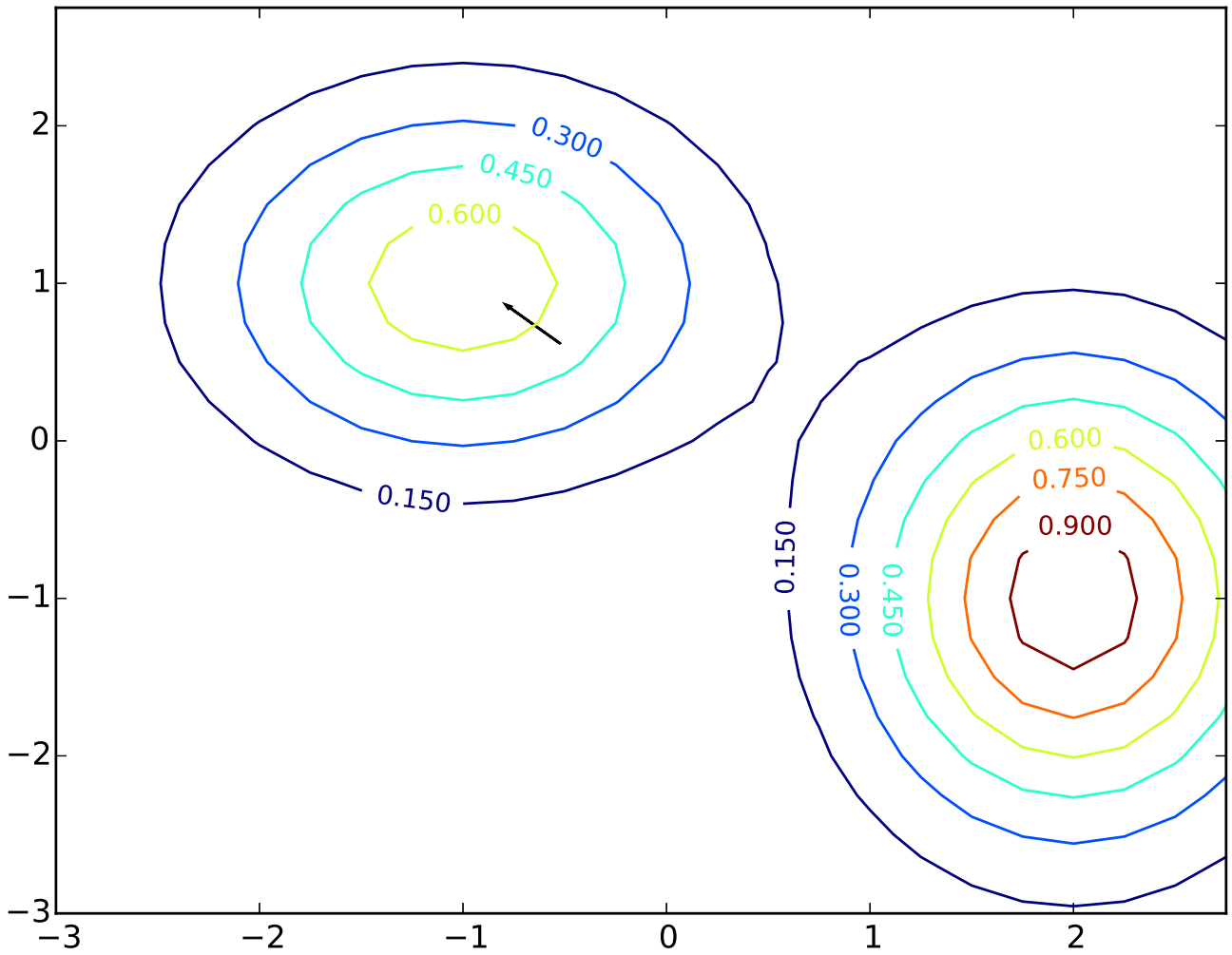
gradient ascent example (2)



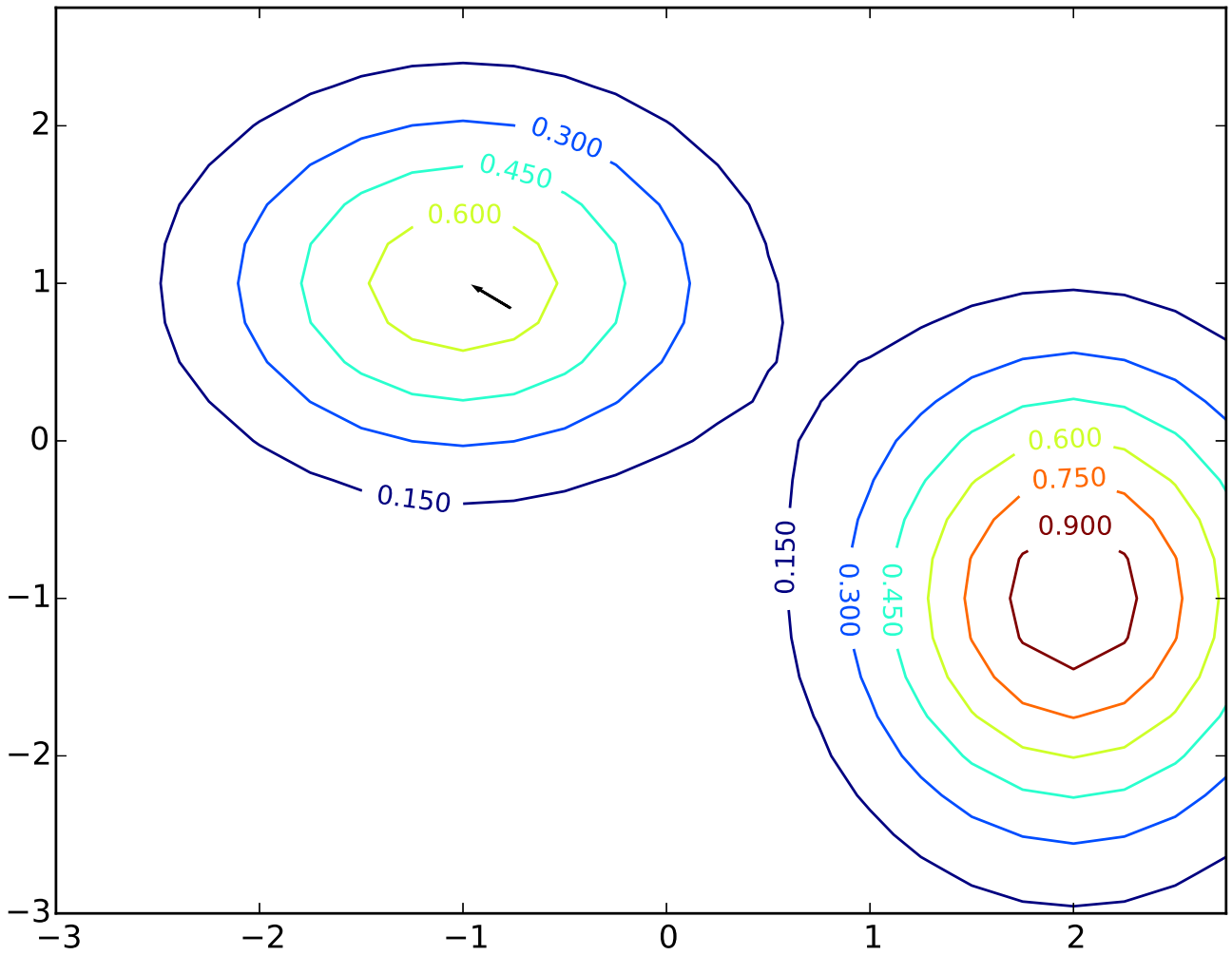
gradient ascent example (2)



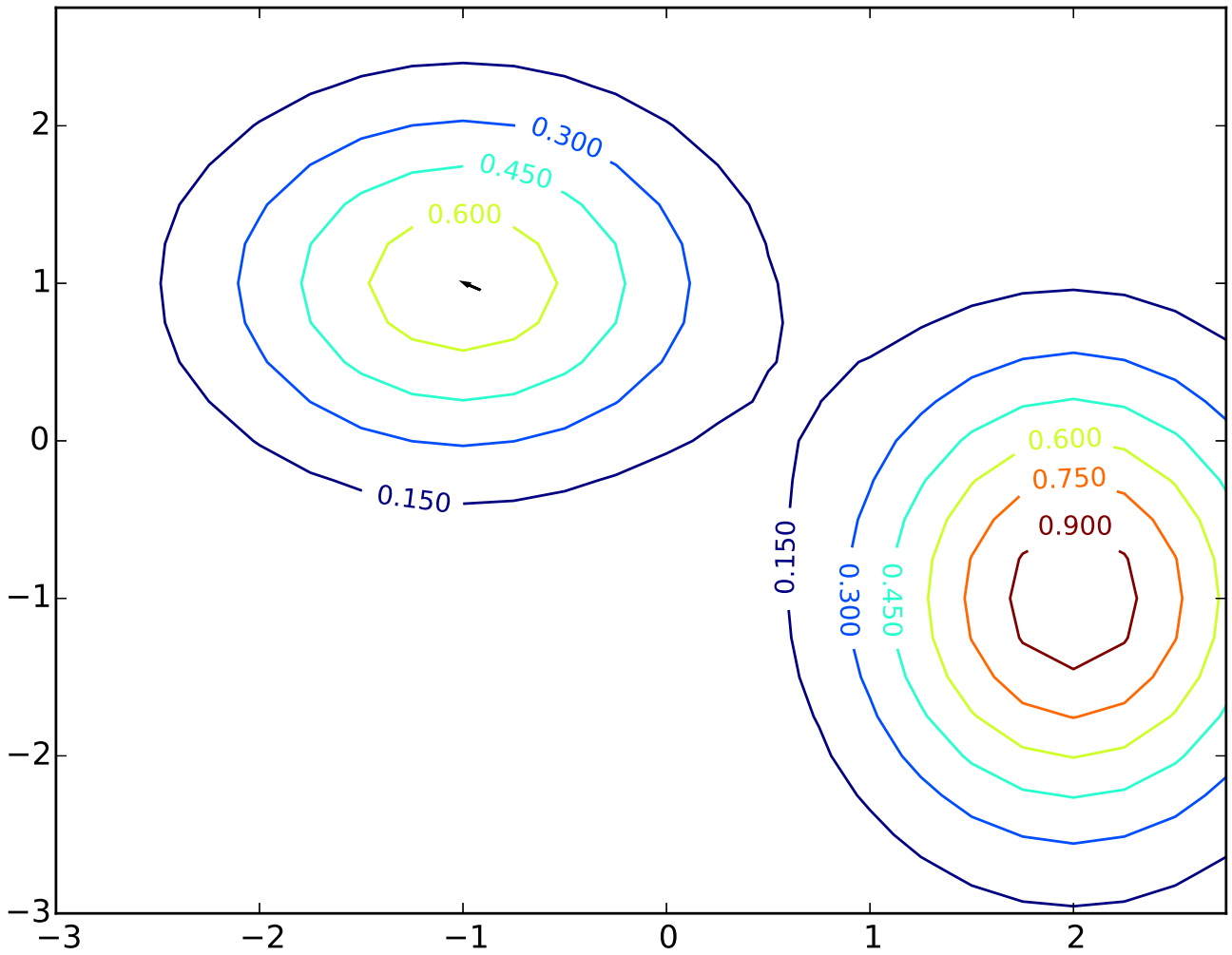
gradient ascent example (2)



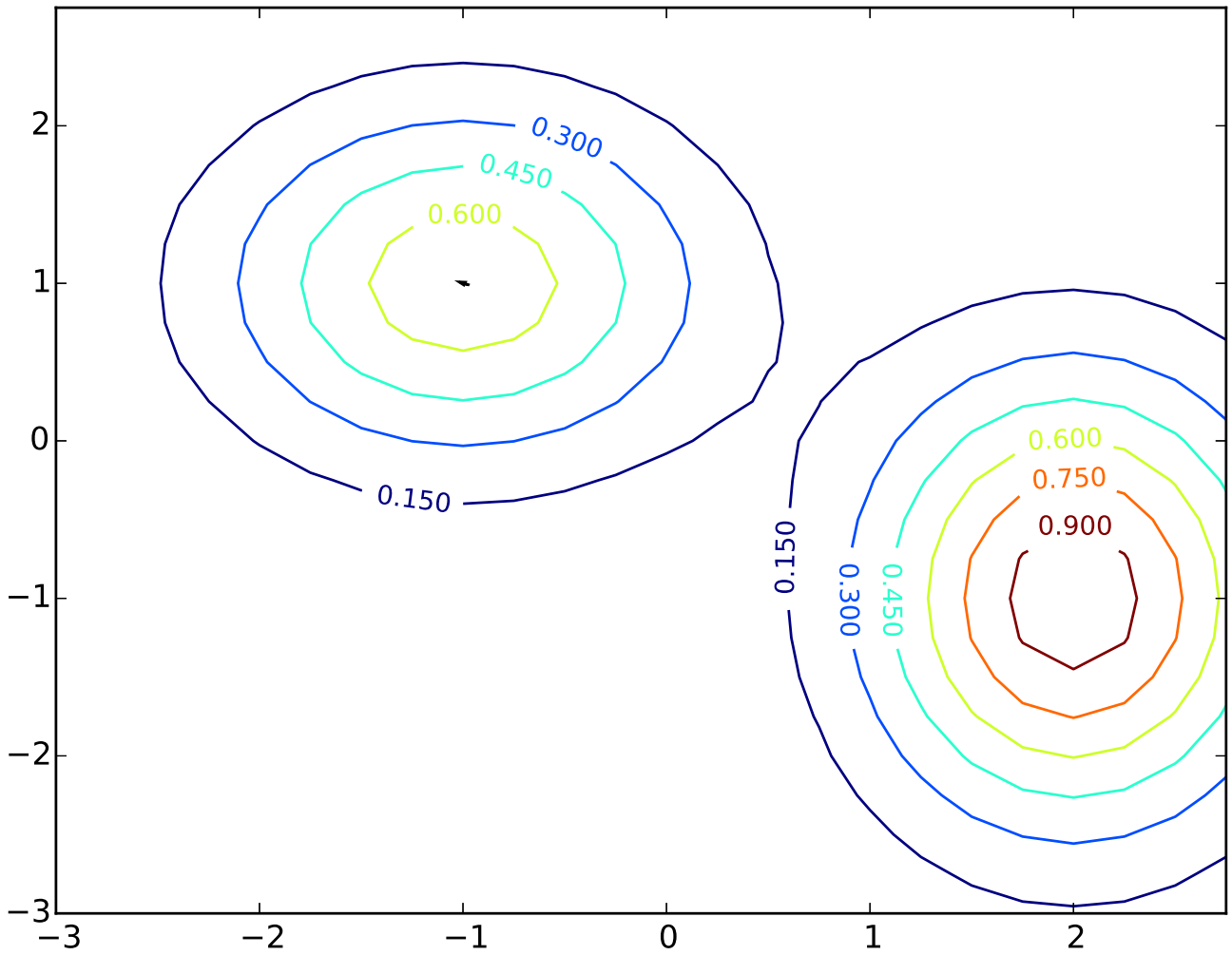
gradient ascent example (2)



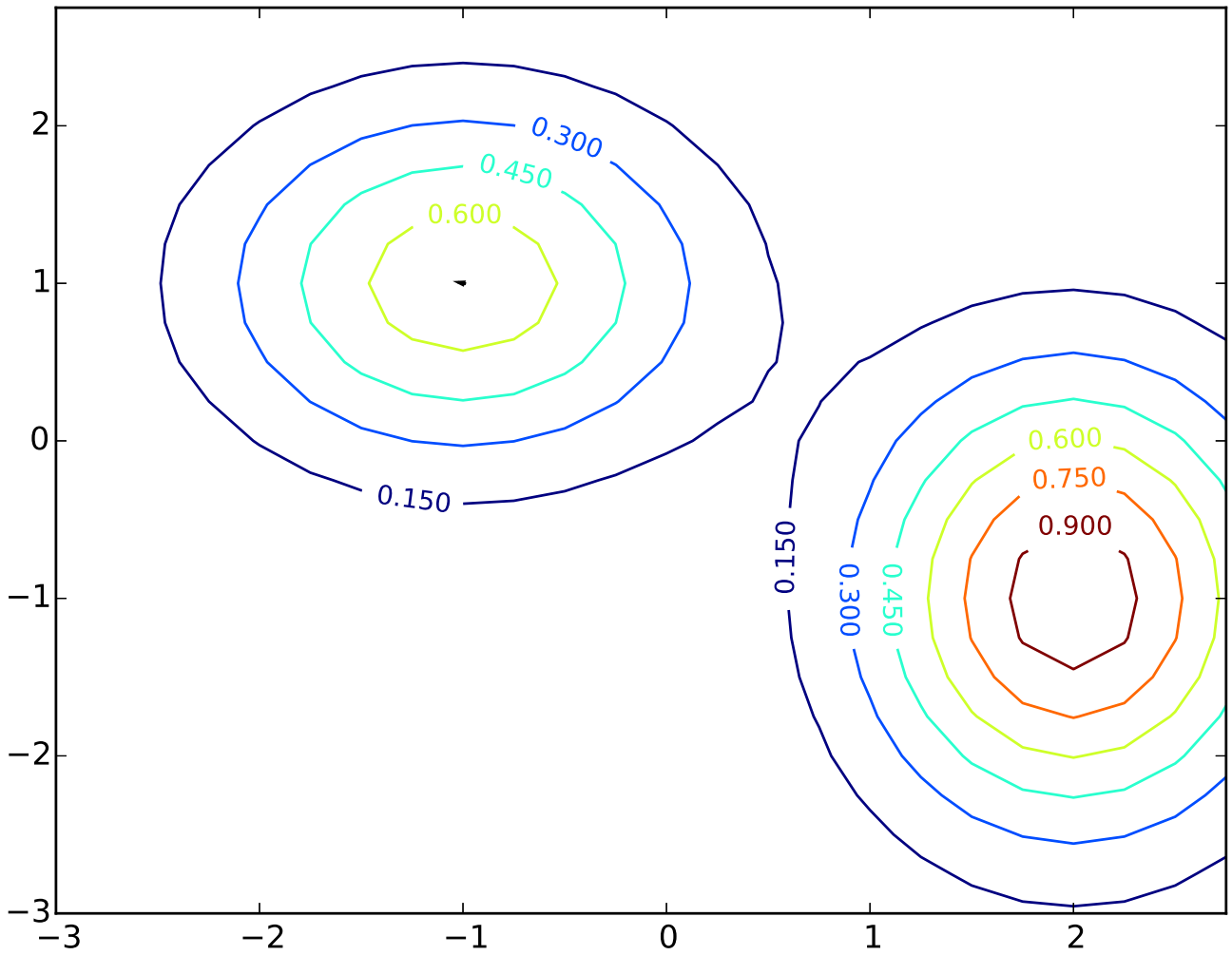
gradient ascent example (2)



gradient ascent example (2)

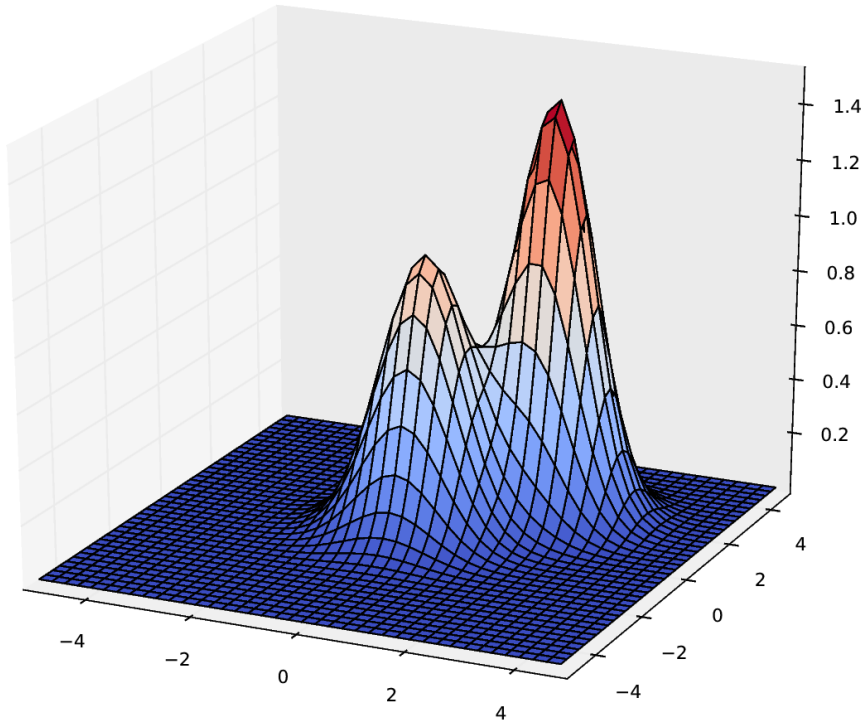


gradient ascent example (2)



local and global maxima/minima

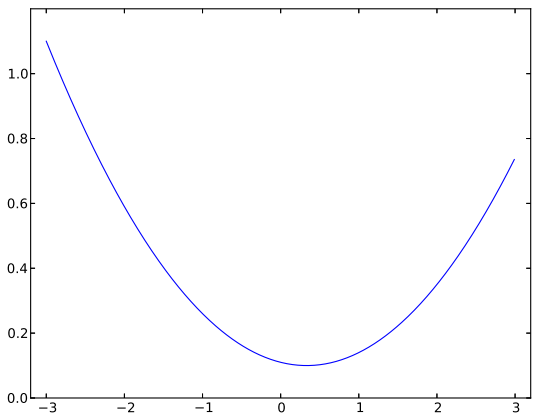
- some functions have **local maxima or minima**

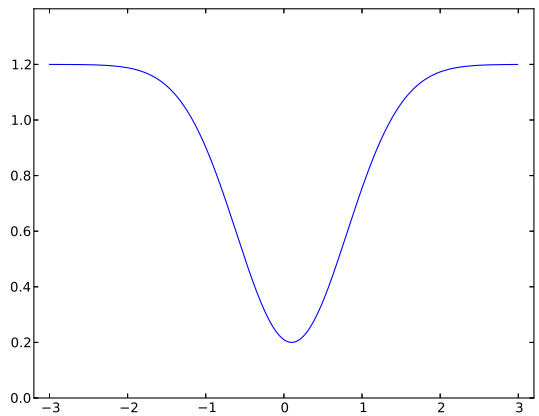


- these functions are harder to optimize because the local (but not global) optima also have a gradient of 0

convex and concave functions

- a function is **convex** if it always curves downwards
- equivalently, if we draw a line between two points of the surface, the surface is always below the line



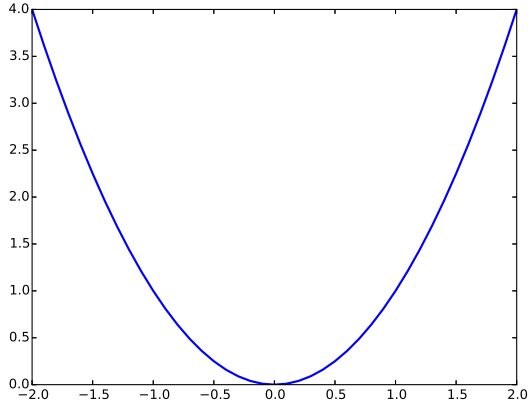


- the point of this: if we find a local optimum (gradient is 0) of a convex function, this is **guaranteed to be the minimum**
- conversely, a function is **concave** if it always curves upwards

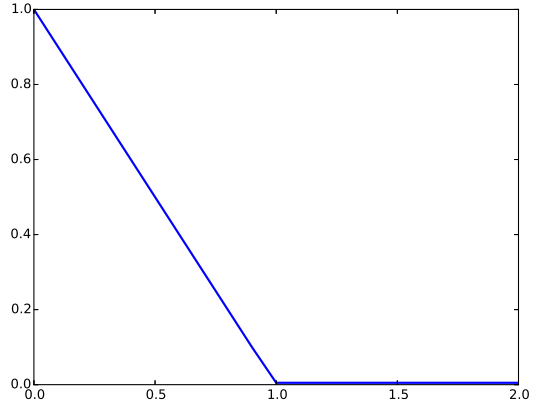
stochastic gradient descent

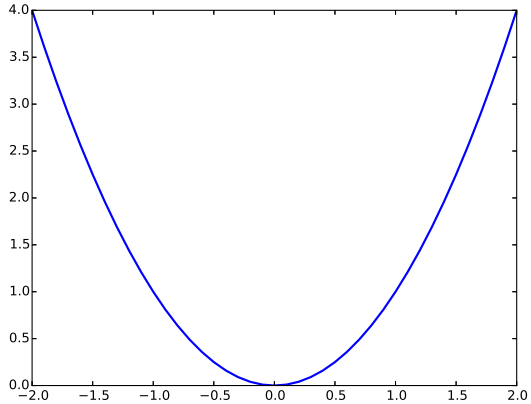
- in some cases it is cumbersome to compute the gradient
  - because it depends on all the data in the training set
- **stochastic gradient descent**: simplify the computation by computing the gradient using just a small part
  - typically, a single training example
- pseudocode:
  1. set w to some initial value, and select a suitable step size c
  2. select a single training instance x
  3. compute the gradient $\nabla f(w)$ **using x only**
  4. if we are "done", stop
  5. otherwise, subtract $c \cdot \nabla f(w)$ from w and go back to step 2
- (stopping criterion shouldn't be based on just a single instance)

SVM and LR have convex objective functions

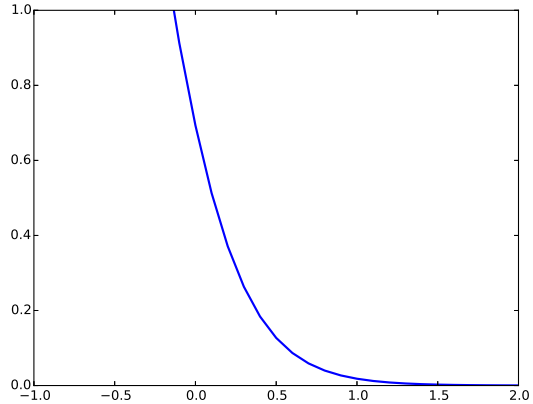


+





+



optimizing SVM and LR

- since the objective functions of SVM and LR are convex, we can find $\boldsymbol{w}$ by stochastic gradient descent
- pseudocode:
  - set $\boldsymbol{w}$ to some initial value, e.g. all zero
  - iterate a fixed number of times:
    - select a single training instance $\boldsymbol{x}$
    - select a "suitable" step length η
    - compute the gradient of the hinge loss or log loss
    - subtract step length · gradient from $\boldsymbol{w}$
- note the similarity to the perceptron!

overview

logistic regression

support vector machines

basic optimization

practical information

some comments about assignment 2

- implement SVM and LR and test them in a document classifier
- we'll use the **Pegasos** algorithm – see assignment page
- Pegasos works in an iterative fashion similar to the perceptron
  - . . . so if you start from my perceptron code this will be a breeze
- optional tasks to speed up the implementation using sparse vectors

some clarifications about the paper

- the important part of the paper is the pseudocode in Figure 1
- Pegasos adapts the step length η over time: long steps in the beginning, smaller in the end
- $\langle \boldsymbol{w}, \boldsymbol{x} \rangle$ is the dot product $\boldsymbol{w} \cdot \boldsymbol{x}$
- S is the training set, $|S|$ is the size of S
- T is the number of steps in the algorithm.
  - this is a bit different from our perceptron, where we specified the number of times to process the whole training set.
- the optional line is there for theoretical reasons and can be ignored
- a **subgradient** is a gradient for "abrupt" functions such as the hinge loss

practical information

- solve the assignment individually
- two lab sessions next week
- deadline one week later: **September 24**

seminar tomorrow

- Peter will present a classical paper about document polarity classification
- I'll present some additional material

lecture next week

- on Friday – so no seminar next week
- neural networks:
  - **non-linear** classifiers
  - they are also implemented using objective functions and optimization