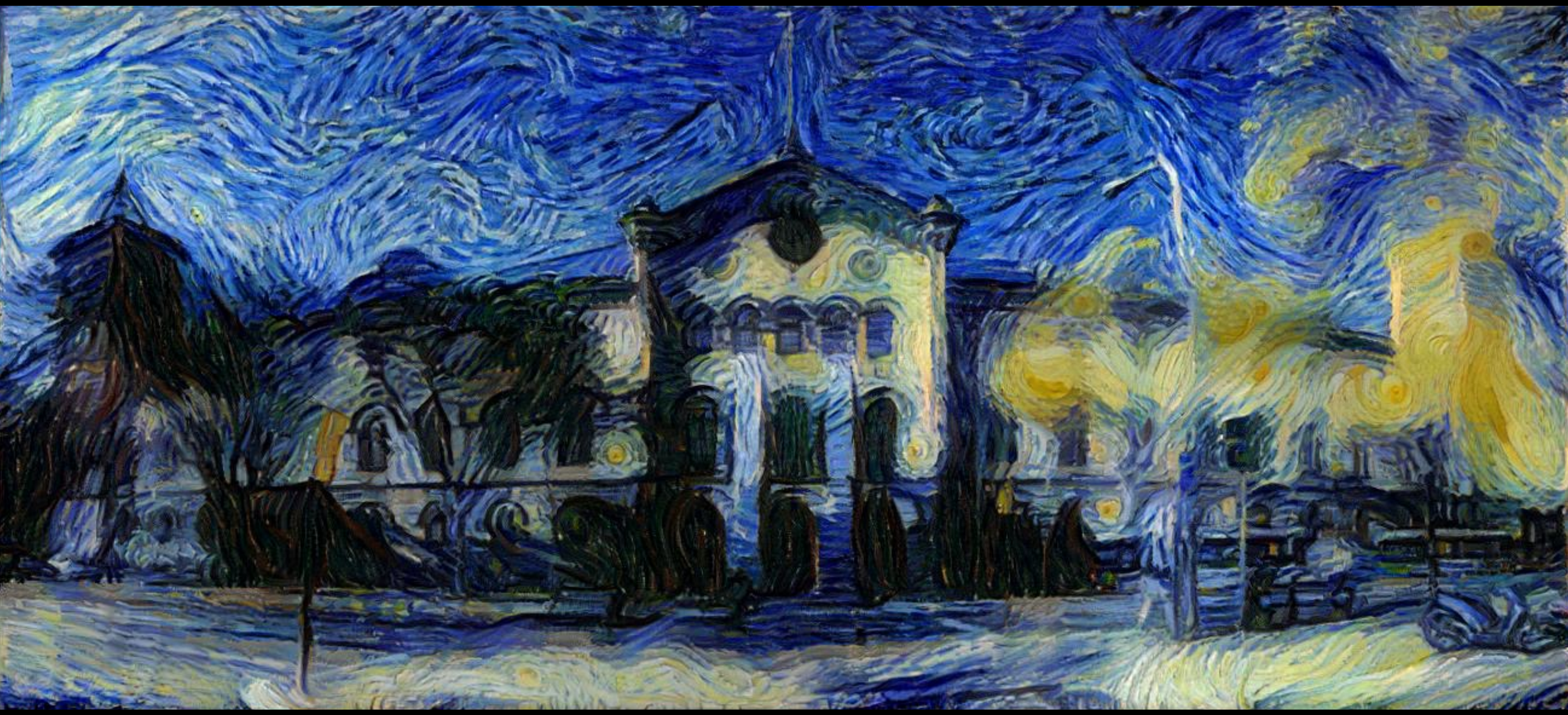


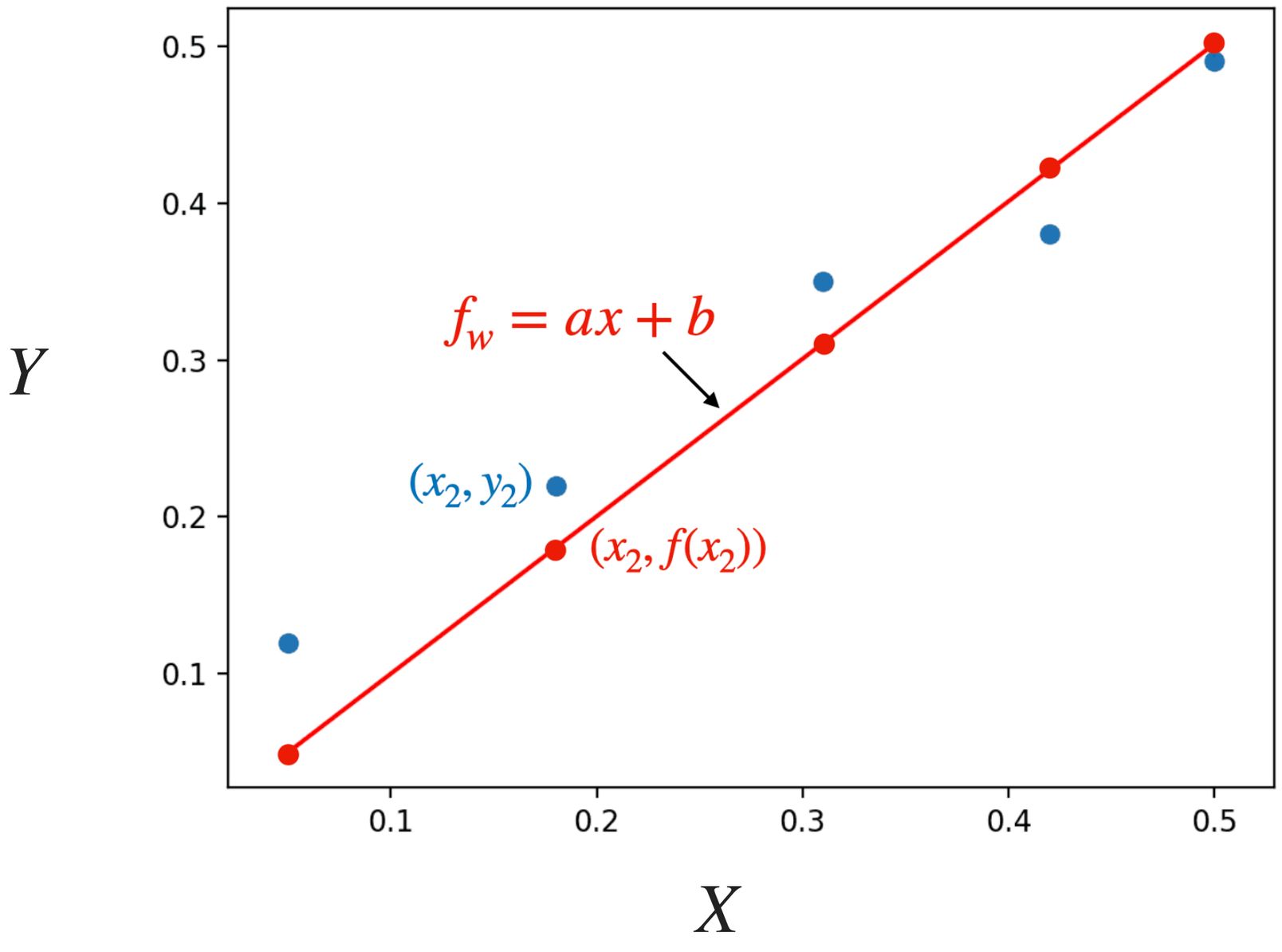
Deep Learning

ML and Optimization

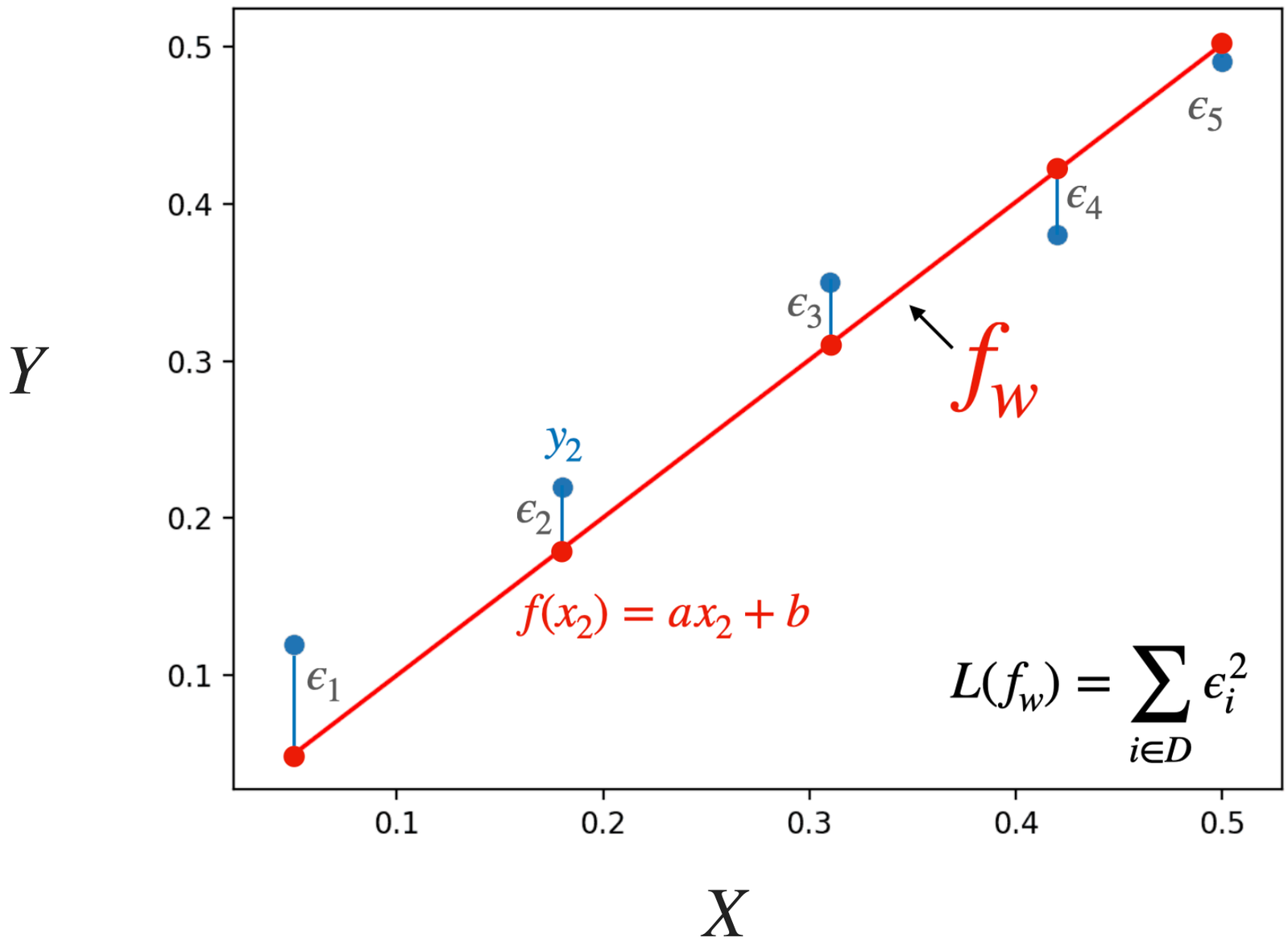
ML Ingredients

- A **dataset** (in most of the cases cosisting of an input X and expected output Y).
- A **model** f_w. It can be thought of as a **function** that accepts your X and returns your $\hat{Y}$ (predicted output).
- Every model has **parameters** w, variables that help define a unique model, and whose values are estimated as a result of learning from data.
- **Cost/Loss function** L. When **optimized**, it makes the predictions of the ML algorithm *estimate* the *actual* values to the best of its ability. The optimization of the cost function is the process of **learning**.

ML Ingredients



ML Ingredients



Optimization

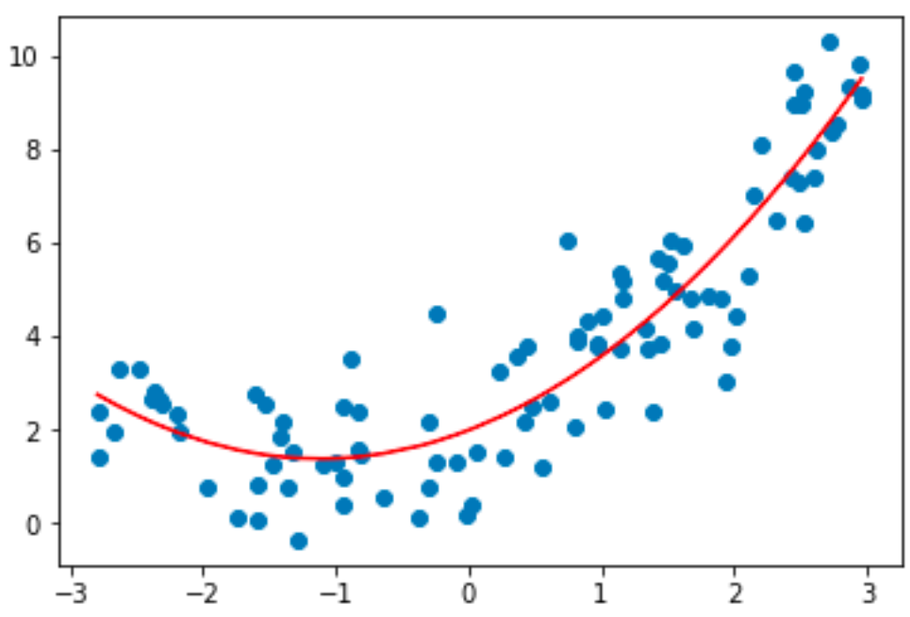
Mathematical optimization is the selection of a best element, with regard to some criterion, from some set of available alternatives.

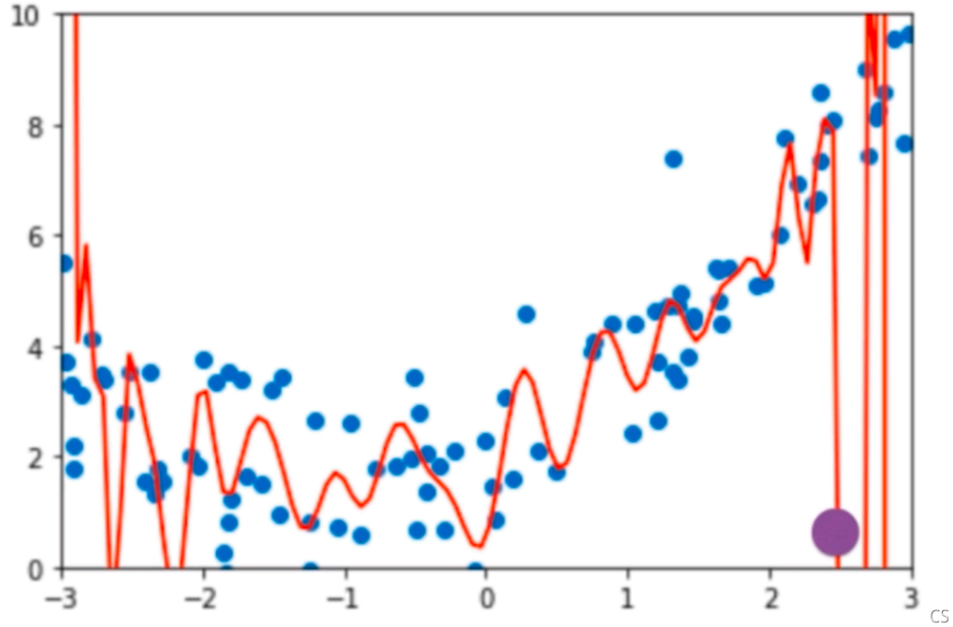
> In the simplest case, an optimization problem consists of maximizing or **minimizing a real function by systematically choosing input values from within an allowed set and computing the value of the function**.
>
> But we need more efficient techniques...

Optimization

NOTE: Although optimization provides a way to minimize the **loss function** for ML, in essence, **the goals of optimization and learning are fundamentally different**.

The former is primarily concerned with **minimizing an objective** whereas the latter is concerned with finding a **suitable model for generalization**, given a finite amount of data.

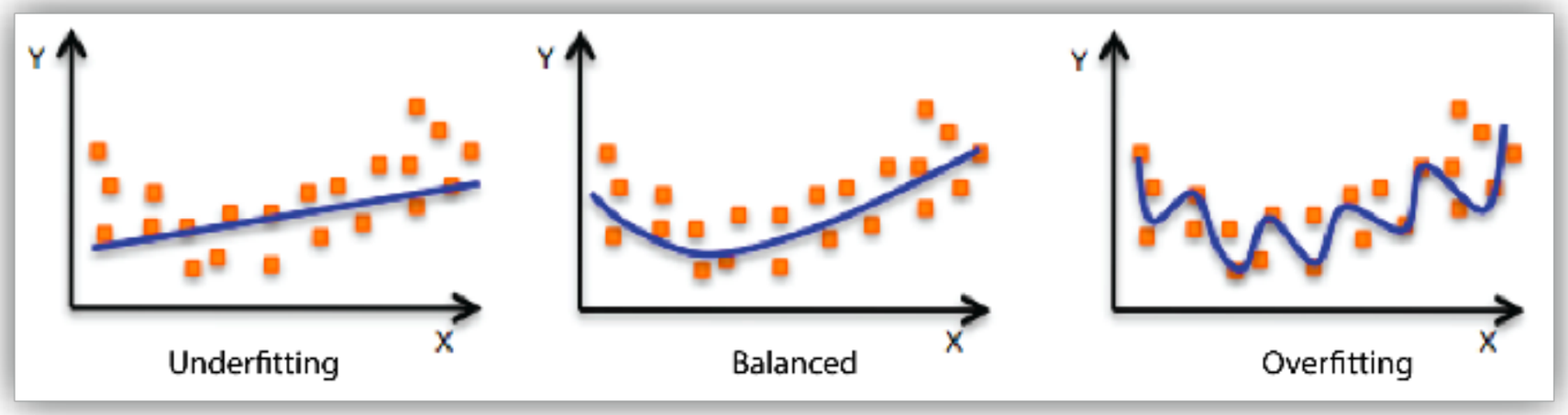




Training (approximation) error and **generalization** error generally differ.

Optimization

Optimization materializes in Machine Learning by **minimizing an objective function such as a function that penalizes for mistakes of the model** while controlling for **overfitting**.



We will talk here about **local methods** that are characterized by the (iterative) search of an optimal value within a neighboring set of the parameter space.

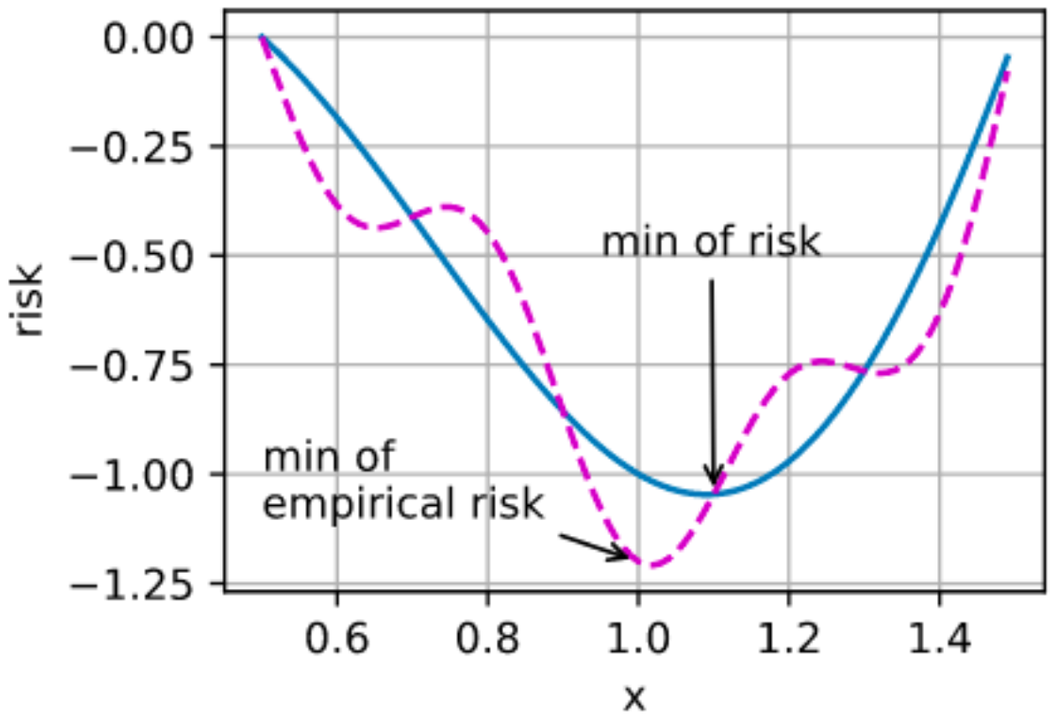
We have a **large variety** of methods that were recently developed.

Empirical Risk Minimization

In order to minimize the **generalization error** of a model we need to pay attention to the **Empirical Risk** concept.

Suppose we have a limited amount of data (the training dataset), but our objective is to process new data (the training dataset is a sample of the population).

The **empirical risk** is an average loss on the training dataset while the **risk** is the expected loss on the entire population of data.



The minimum of the empirical risk on a training dataset may be at a different location from the minimum of the risk (generalization error).

Empirical Risk Minimization

On a supervised setting, we want to find a function or a model $f_\theta(\,.\,)$ that describes the relationship between a random feature vector $\mathbf{x}$ and the label target vector y. We assume a joint distribution $p_{data}(\mathbf{x}, y)$.

	$\mathbf{x}$							y
	Name	Team	Number	Position	Age	Height	Weight	College	Salary
0	Avery Bradley	Boston Celtics	0.0	PG	25.0	6-2	180.0	Texas	7730337.0
1	Jae Crowder	Boston Celtics	99.0	SF	25.0	6-6	235.0	Marquette	6796117.0
2	John Holland	Boston Celtics	30.0	SG	27.0	6-5	205.0	Boston University	NaN
3	R.J. Hunter	Boston Celtics	28.0	SG	22.0	6-5	185.0	Georgia State	1148640.0
4	Jonas Jerebko	Boston Celtics	8.0	PF	29.0	6-10	231.0	NaN	5000000.0
5	Amir Johnson	Boston Celtics	90.0	PF	29.0	6-9	240.0	NaN	12000000.0
6	Jordan Mickey	Boston Celtics	55.0	PF	21.0	6-8	235.0	LSU	1170960.0
7	Kelly Olynyk	Boston Celtics	41.0	C	25.0	7-0	238.0	Gonzaga	2165160.0
8	Terry Rozier	Boston Celtics	12.0	PG	22.0	6-2	190.0	Louisville	1824360.0
9	Marcus Smart	Boston Celtics	36.0	PG	22.0	6-4	220.0	Oklahoma State	3431040.0
10	Jared Sullinger	Boston Celtics	7.0	C	24.0	6-9	260.0	Ohio State	2569260.0

Note: non numerical features must be transformed before training!

Empirical Risk Minimization

We start by defining a loss function L, evaluated as $L(f(\mathbf{x}), y)$ that gives us a penalization for the difference between predictions $f(\mathbf{x})$ and the true label y.

$$\text{For example, } L(f_w(\mathbf{x}), y)) = (y - f_w(\mathbf{x}))^2$$

Now, taking the expectation of the loss we have our risk R (that we want to minimize):

$$R(f) = \mathop{\mathbb{E}}_{\mathbf{x},y\sim p_{data}}[L(f_w\mathbf{x}), y)] = \int L(f_w(\mathbf{x}), y) dp_{data}(\mathbf{x}, y)$$

Empirical Risk Minimization

However, we don't know $p_{data}(\mathbf{x}, y)$, we only have access to a sample trining set $\mathscr{D} = (\mathbf{x}_i, y_i) \sim p_{data}$.

Therefore, we can approximate the risk with the **empirical risk**:

$$R_{emp}(f) = \frac{1}{n} \sum_{i=1}^{n} L(f_w(\mathbf{x}_i), y_i)$$

The **Empirical Risk Minimization** (ERM) principle says that our learning algorithm should minimize the empirical risk.

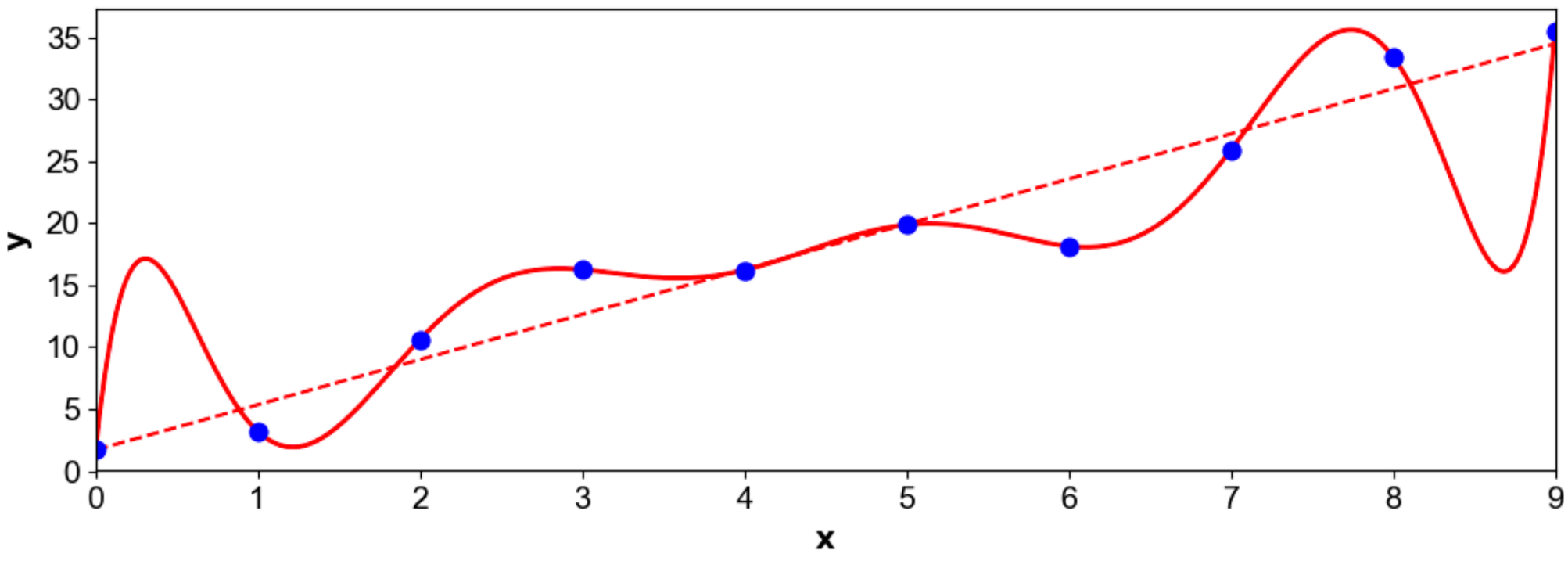
Our interest here in terms of optimization is:

$$w^* = \arg\min_{w} R_{emp}(f); w \in \mathbb{R}^n$$

Empirical Risk Minimization

In order to obtain a good generalization level we have to follow a **good training methodology**.

Keywords: over/under-fitting, cross-validation, etc.



Empirical Risk Minimization

The solution to the overfitting problem is to use an appropriate learning methodology during optimization: **cross-validation**.

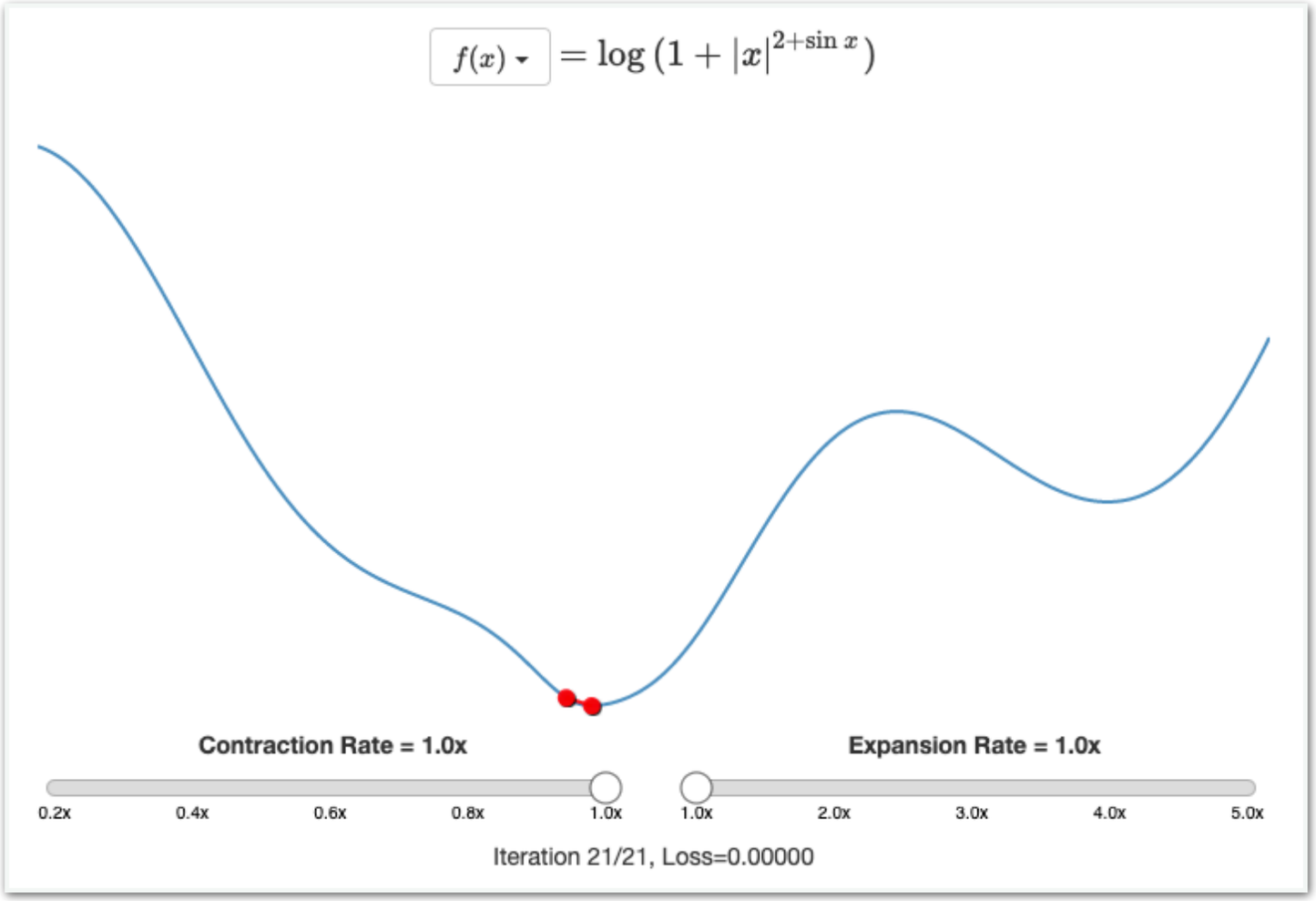
This consists of dividing the data set into two complementary subsets, evaluating the model on a subset (called **training set)**, and validating it on the other subset (called **test set**).

The model is only fit with the training data set and from there it calculates the output values for the test data set (values that it has not analyzed before).

If the result on the test data is not good enough, we have to change the model and look for a simpler one!

Gradient-free optimization: Nelder-Mead

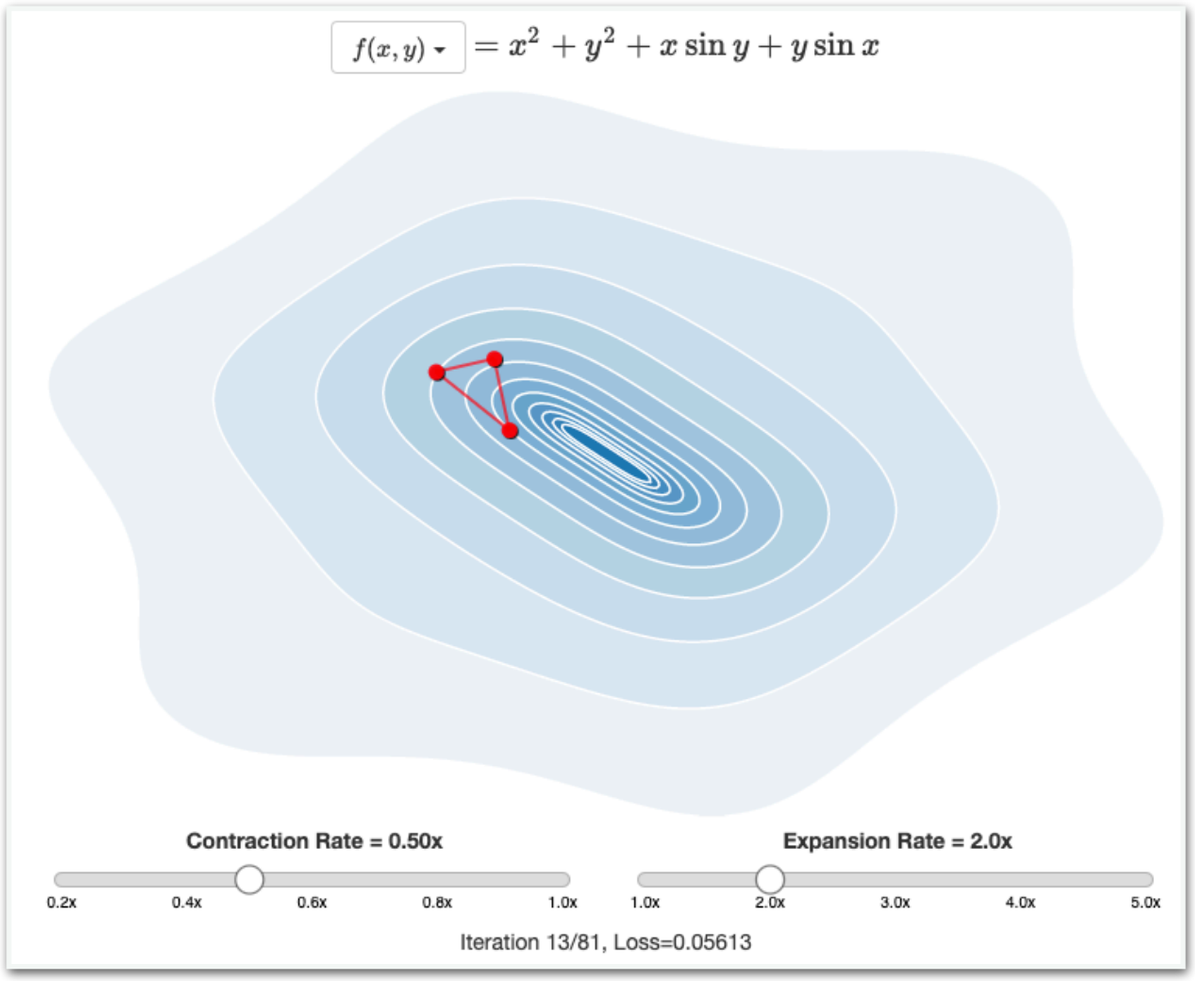
The most simple thing we can try to minimize a function would be to sample two points relatively near each other, and and just repeatedly take a step down away from the largest value.



http://www.benfrederickson.com/numerical-optimization/

Gradient-free optimization: Nelder-Mead

The most simple thing we can try to minimize a function would be to sample two points relatively near each other, and and just repeatedly take a step down away from the largest value.



http://www.benfrederickson.com/numerical-optimization/

Gradient Descend

The Nelder-Mead method can be easily extended into higher dimensional examples, all that's required is taking one more point than there are dimensions.

In the case of deep learning and other high dimensional problems the main limitation of this method is **the number of function evaluations** wee need.

Let’s see an alternative: the use of **function derivatives**.

Let's suppose that we have a function we are considering the minimization of a function:

$$f(x) = x^2$$

Our objective is to find the argument x that **minimizes** this function.

Gradient Descend

Derivative definition:

The sign of the derivative is + if $f(x+h) > f(x)$

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

The derivative specifies how to scale a small change in the input in order to obtain the corresponding change in the output:

$$f(x+h) \approx f(x) + hf'(x)$$

Numerical derivatives

It can be shown that the "centered difference formula" is better when computing numerical derivatives:

$$\lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$$

The error in the "finite difference" approximation can be derived from Taylor's theorem and, assuming that f is differentiable, is $O(h)$. In the case of "centered difference" the error is $O(h^2)$.

Gradient Descend

The derivative tells how to change x in order to make a small improvement in f.

Then, we can follow these steps to decrease the value of the function (minimize):

- Start from a random x
- Compute the derivative $f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x-h)}{2h}$
- Walk a small step (possibly weighted by the derivative module) in the **opposite** direction of the derivative, because we know that $f(x - h \text{ sign}(f'(x))$ is less than $f(x)$ for a small step.

The search for the minima ends when the derivative is zero (very small) because we have no more information about which direction to move.

Algorithm The general gradient descent algorithm.

Input: initial weights $\theta^{(0)}$, iterations T, learning rate η
Output: final weights $\theta^{(T)}$

1. **for** $t = 0$ **to** $T - 1$
2. compute $\nabla L(\theta^{(t)})$
3. $\theta^{(t+1)} := \theta^{(t)} - \eta \nabla L(\theta^{(t)})$
4. **return** $\theta^{(T)}$

Gradient Descend

There are two problems with numerical derivatives:

- They are approximate.
- They are slower than necessary to evaluate (we need **two evaluations** of one-dimensional functions).

What about using calculus?

- Start from a random x
- Compute the derivative $f'(x)$ analitically
- Walk a small step in the **opposite** direction of the derivative.

Gradient Descend

Let's consider a n-dimensional function $f : \mathbb{R}^n \to \mathbb{R}$.

$$f(\mathbf{x}) = \sum_n x_n^2$$

Our objective is to find the argument that minimizes this function.
The gradient, ∇f, is the vector whose components are the n partial derivatives of f. It is thus a vector-valued function:

$$\nabla f = (\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n})$$

The gradient points in the direction of the greatest rate of **increase** of the function.

> It is important to be aware that this gradient computation is very expensive: if $\mathbf{x}$ has dimension n, we have to evaluate f at $2 * n$ points (2 for every dimension).

Gradient Descend

```
def fin_dif_partial_centered(x,
                             f,
                             i,
                             h=1e-6):
    '''
    This method returns the partial derivative of the i-th component of f at x
    by using the centered finite difference method
    '''
    w1 = [x_j + (h if j==i else 0) for j, x_j in enumerate(x)]
    w2 = [x_j - (h if j==i else 0) for j, x_j in enumerate(x)]
    return (f(w1) - f(w2))/(2*h)

def gradient_centered(x,
                      f,
                      h=1e-6):
    '''
    This method returns the gradient vector of f at x
    by using the centered finite difference method
    '''
    return[round(fin_dif_partial_centered(x,f,i,h), 10) for i,_ in enumerate(x)]

def f(x):
    return sum(x_i**2 for x_i in x)

x = [1.0,1.0,1.0]
gradient_centered(x,f)

>>> 3.000000 [2.0000000001, 2.0000000001, 2.0000000001]
```

Gradient Descend

The step size, h, is a slippy concept: if it is too small we will slowly converge to the solution, if it is too large we can diverge from the solution.

There are several policies to follow when selecting the step size:

- Constant size steps. In this case, the size step determines the precision of the solution.
- Decreasing step sizes.
- At each step, select the optimal step.

Magical step size: 0.01

Gradient Descend and Machine Learning

In general, we have:

- A dataset $(\mathbf{x}, y)$ of N examples.
- A target function $f_{\mathbf{w}}$, that we want to minimize, representing the mean **discrepancy between our data points and the model predictions** . The model is indexed by a set of parameters $\mathbf{w}$.
- The gradient of the target function, g_f.

Let's suppose that our problem (i.e. regression) is to find the optimal parameters $\mathbf{w}$ that minimize the following expression:

$$\frac{1}{N}\sum_{i}(y_i - f(\mathbf{x}_i, \mathbf{w}))^2$$

Example

And let's suppose that we choose our model to be a one-dimensional linear model:

$$f(\mathbf{x}, \mathbf{w}) = w \cdot x$$

We can implement **gradient descend** in the following way:

Dataset

Loss function

Gradient of the loss function

Weighted Step

```
import numpy as np
import random

# f = 2x
x = np.arange(10)
y = np.array([2*i for i in x])

# f_target = 1/n Sum (y - wx)**2
def target_f(x,y,w):
    return np.sum((y - x * w)**2.0) / x.size

# gradient_f = 2/n Sum 2wx**2 - 2xy
def gradient_f(x,y,w):
    return 2 * np.sum(2*w*(x**2) - 2*x*y) / x.size

def step(w,grad,alpha):
    return w - alpha * grad
```

Example

```
def BGD(target_f,
        gradient_f,
        x,
        y,
        toler = 1e-6,
        alpha=0.01):
    '''
    Batch gradient descend by using a given step
    '''
    w = random.random()
    val = target_f(x,y,w)
    i = 0
    while True:
        i += 1
        gradient = gradient_f(x,y,w)
        next_w = step(w, gradient, alpha)
        next_val = target_f(x,y,next_w)
        if (abs(val - next_val) < toler):
            return w
        else:
            w, val = next_w, next_val

BGD(target_f, gradient_f, x, y)
>>> 2.000093
```

It is called (batch) gradient descend because at every step, when computing

`next_val = target_f(x,y,next_w)`

we are using **the whole dataset**!

Stochastic Gradient Descend

The last function evals the whole dataset $(\mathbf{x}_i, y_i)$ at every step. If the dataset is large, this strategy is too costly.

In this case we will use a strategy called **SGD** (*Stochastic Gradient Descend*), that is based on the following fact:

When learning from data, the cost function **is additive**: it is computed by adding sample reconstruction errors.
In this case, it can be shown that we can compute a good estimate of the gradient (and move towards the minimum) by using only **one data sample** (or a small data sample) at each iteration.

If we apply this method we have some **theoretical guarantees** to find a good minimum.

- Kiwiel, Krzysztof C. (2001). "Convergence and efficiency of subgradient methods for quasiconvex minimization". Mathematical Programming, Series A. 90 (1). Berlin, Heidelberg: Springer. pp. 1–25.
- Bottou, Léon (1998). "Online Algorithms and Stochastic Approximations". Online Learning and Neural Networks. Cambridge University Press.

Stochastic Gradient Descend

A full iteration over the dataset is called **epoch**.

```
nb_epochs = 100
for i in range(nb_epochs):
    np.random.shuffle(data)
    for sample in data:
        grad = evaluate_gradient(target_f, sample, w)
        w = w - learning_rate * grad
```

In order to fullfill the guarantees of SGD, at every epoch, data must be processed in a random order.

Loss Functions

$$L(y, f(\mathbf{x})) = \frac{1}{n} \sum_i \ell(y_i, f(\mathbf{x_i}))$$

Loss functions represent the price paid for inaccuracy of predictions in classification/regression problems.

In regression problems, the most common loss function is the **square loss function**:

$$L(y, f(\mathbf{x})) = \frac{1}{n} \sum_i (y_i - f(\mathbf{x}_i))^2$$

We assume there is a function f such that $y_i = f(x_i) + \epsilon_i$, where ϵ_i is Gaussian noise.

In classification this function could be the **zero-one loss**, that is

$$\ell(y_i, f(\mathbf{x_i}))$$

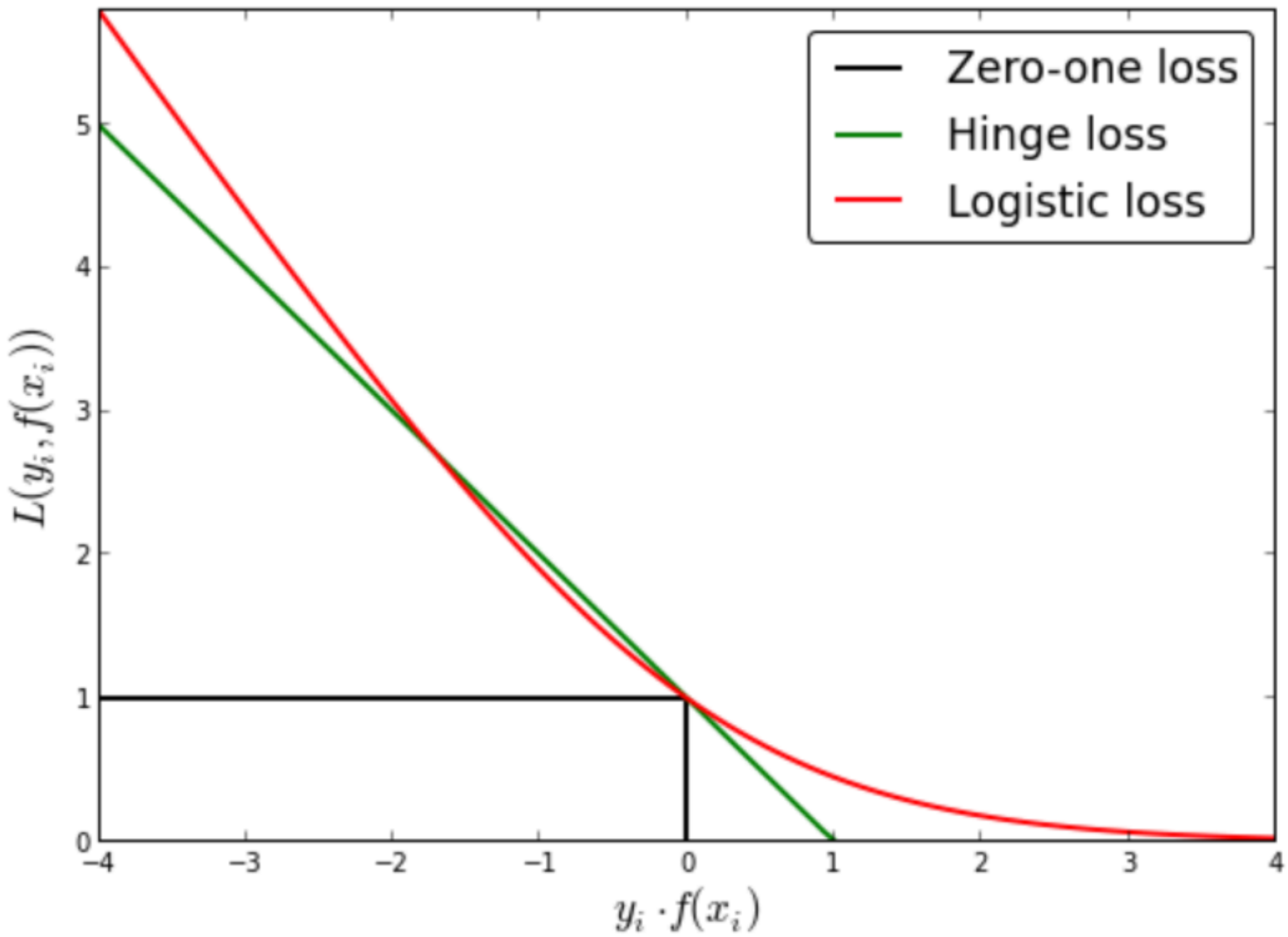
is 0 when $y_i = f(\mathbf{x}_i)$ and 1 otherwise.

This function is discontinuous with flat regions and is thus extremely hard to optimize using gradient-based methods. For this reason it is usual to consider a proxy to the loss called a ***surrogate loss function***. For computational reasons this is usually convex function.

Surrogate Loss Functions

$$L(y, f(\mathbf{x})) = \frac{1}{N} \sum_i \max(0, 1 - y_i f(\mathbf{x}_i))$$

Hinge / Margin Loss (i.e. Suport Vector Machines)



Surrogate Loss Functions

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i} log(1 + exp(-y_i \hat{y}_i))$$

Binary Logistic Regression

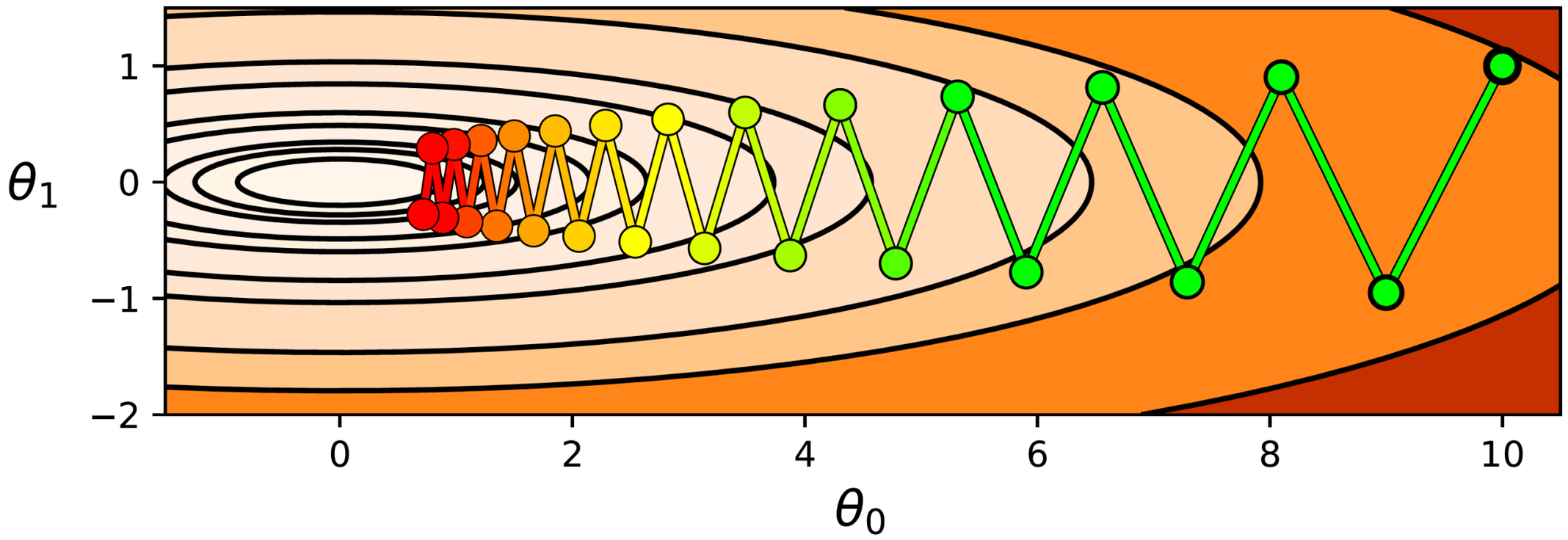
$$L(y, \hat{y}) = -\frac{1}{N} \sum_{i} \sum_{j \in C} y_{i,j} \log \hat{y}_{i,j}$$

Multiclass Cross-Entropy Loss

```
>>> y_true = [[0, 1, 0], [0, 0, 1]]
>>> y_pred = [[0.05, 0.95, 0], [0.1, 0.8, 0.1]]
>>> # Using 'auto'/'sum_over_batch_size' reduction type.
>>> cce = tf.keras.losses.CategoricalCrossentropy()
>>> cce(y_true, y_pred).numpy()
1.177
```

Advanced Gradient Descend

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another, which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum.



Source: Code adapted from Machine Learning Refined. Jeremy Watt and Reza Borhani. 2020. Creative Commons Attribution 4.0 International License.

Advanced Gradient Descend Methods

Momentum is a method to damp out oscillations.

Vanilla gradient descent:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \nabla L(\theta^{(t)})$$

Momentum:

Note that with $\beta = 0$ we recover vanilla Gradient descent.

$$V^{(t+1)} = \underbrace{\beta}_{\text{Constant}} V^{(t)} + \nabla L(\theta^{(t)})$$

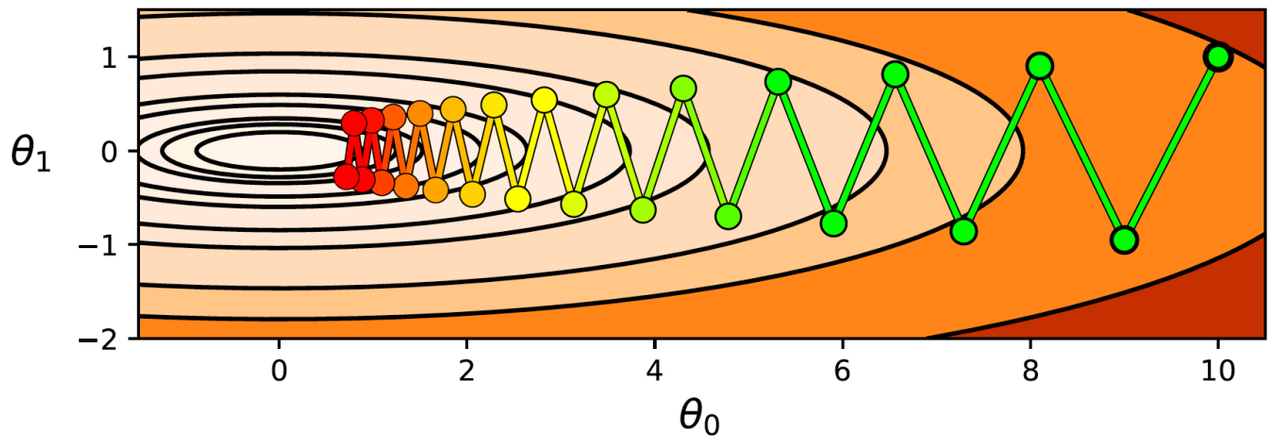
$$\theta^{(t+1)} = \theta^{(t)} - \eta \underbrace{V^{(t+1)}}_{\text{Momentum buffer}}$$

Momentum works by acceleration and smoothing, it makes the trajectories to take more time to react to changes in the loss landscape.

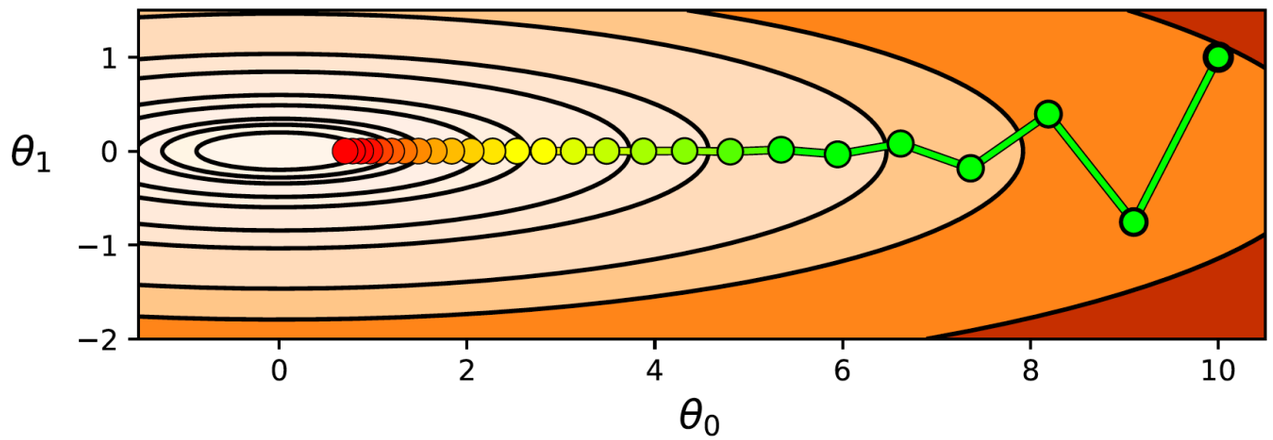
Advanced Gradient Descend

$\mathbf{M}$

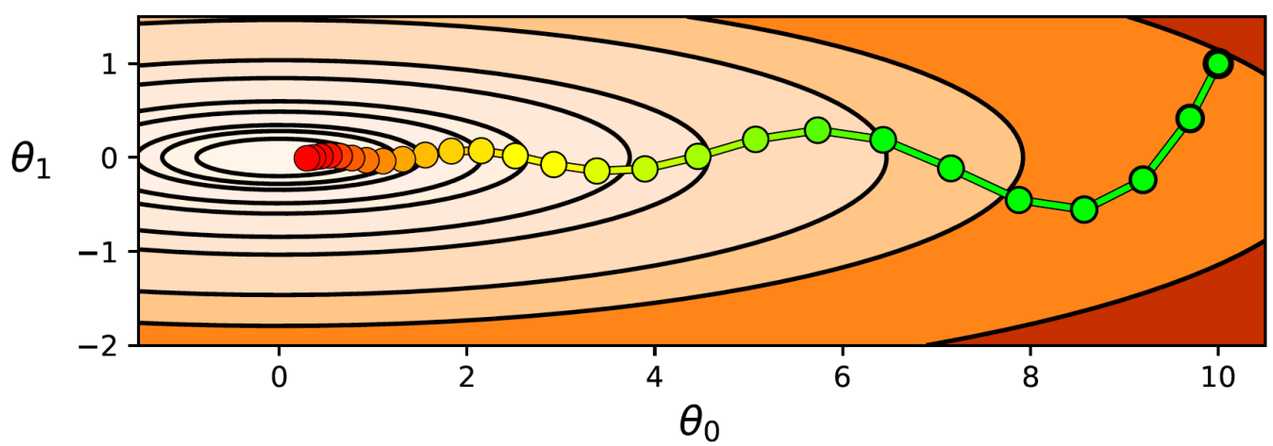
$\beta = 0.0$



$\beta = 0.1$



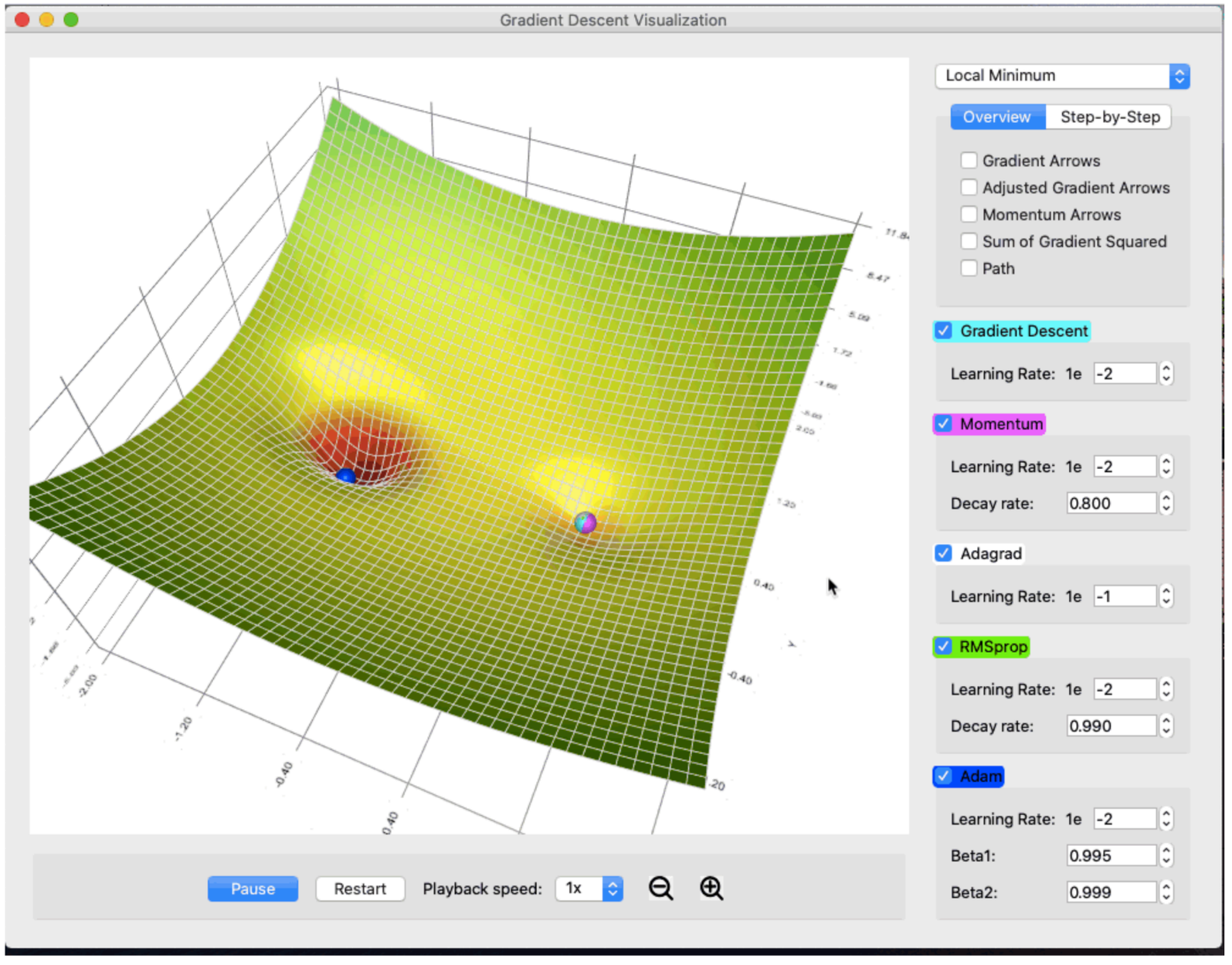
$\beta = 0.7$



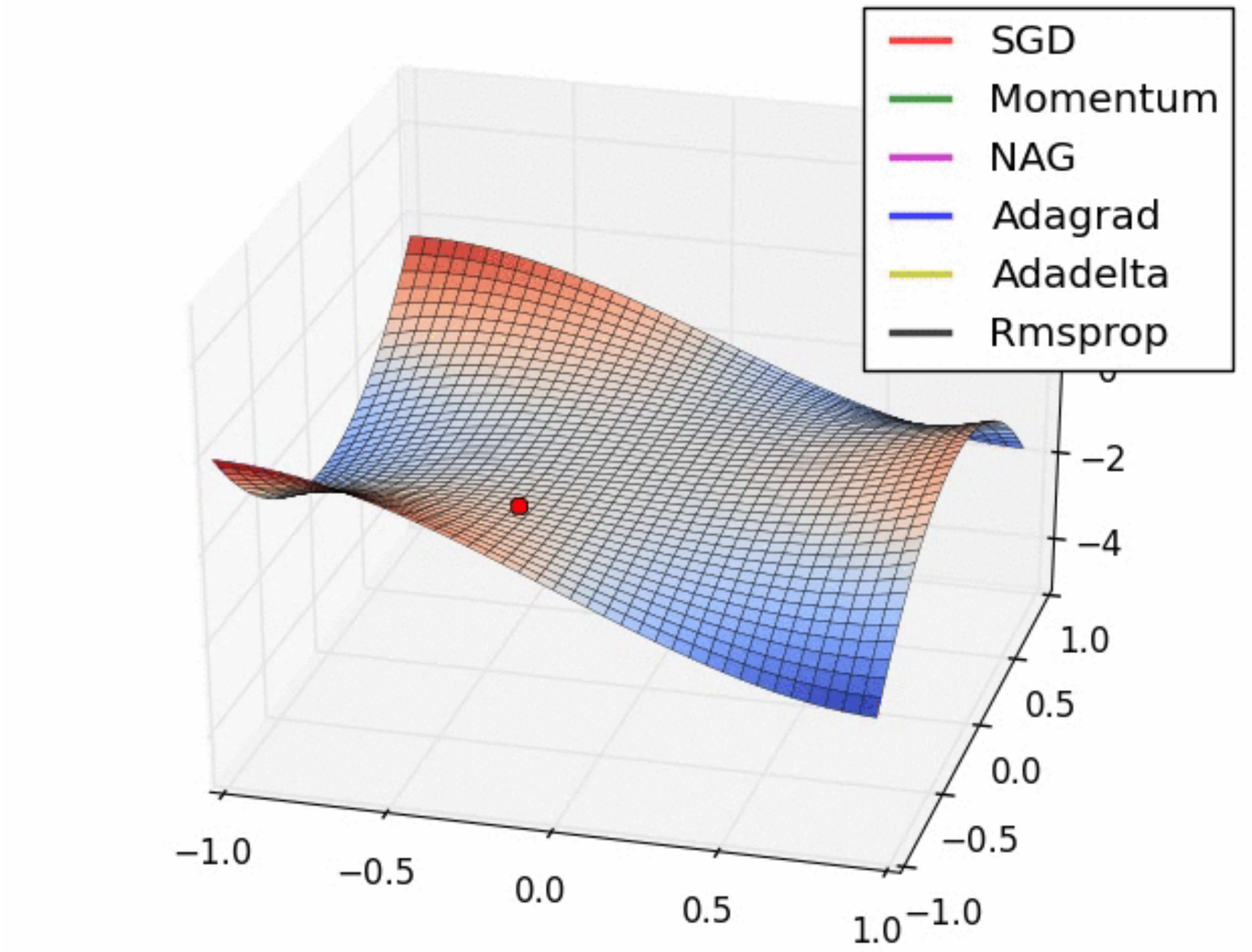
Source: Code adapted from Machine Learning Refined. Jeremy Watt et al. 2020.

Advanced Gradient Descend

Demo from Lili Jiang, from: https://github.com/lilipads/gradient_descent_viz



Advanced Gradient Descend



Notebook

https://colab.research.google.com/github/DeepLearningUB/DeepLearningUB.github.io/blob/master/deep1.ipynb