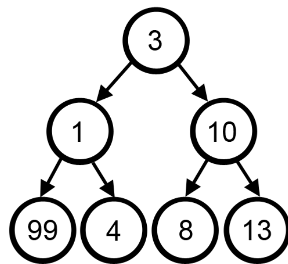
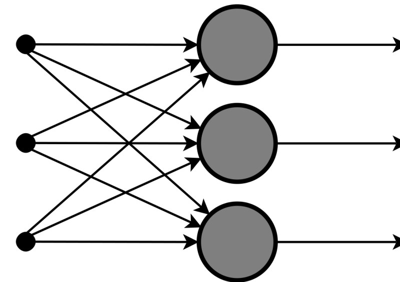
CSE 40171: Artificial Intelligence

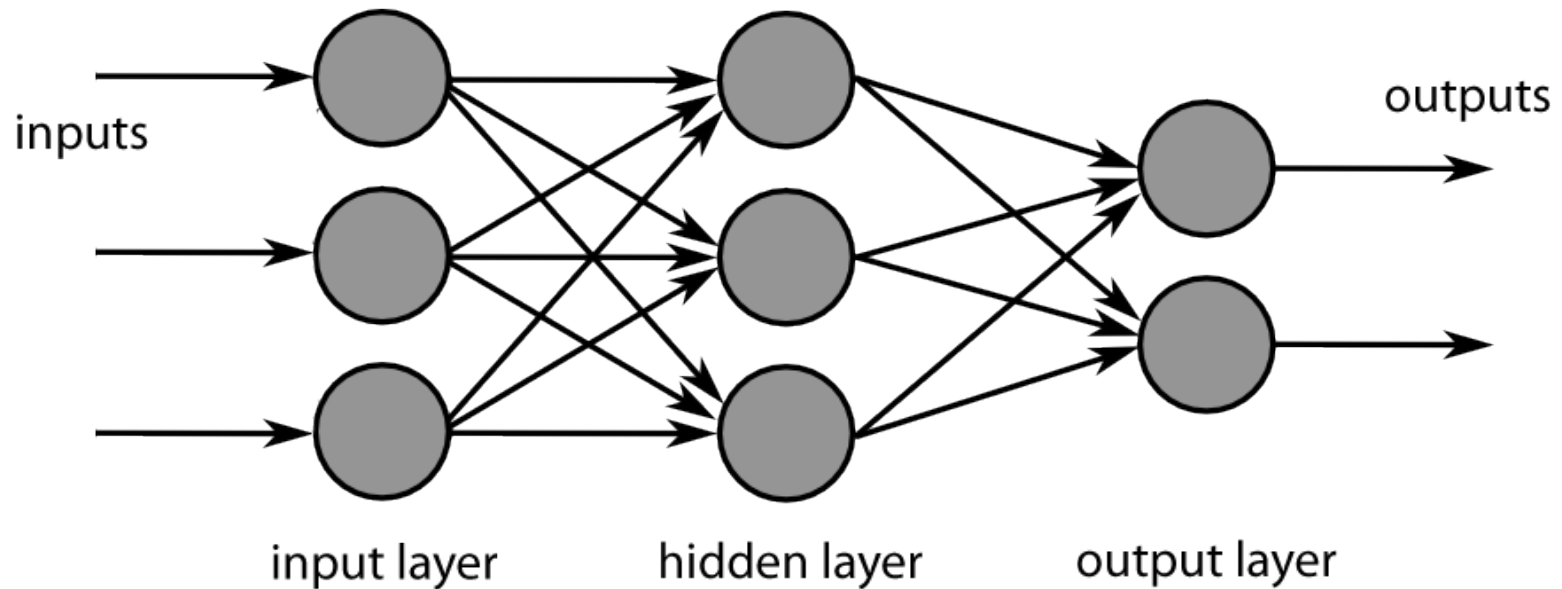




Artificial Neural Networks: Gradient-Based Optimization

Homework #1 has been released
It is due at 11:59PM on 9/16

How do we train a network?



A Neural network with two layers (cc) BY-SA 3.0 Chrislb

Training Loop

1. Draw a batch of training samples **x** and corresponding targets **y**

2. Run the network on **x** (forward pass) to obtain predictions **y_pred**

3. Compute the loss of the network on the batch, a measure of the mismatch between **y_pred** and **y**

4. Update the weights of the network in a way that slightly reduces the loss on this batch

Slide credit: F. Chollet, Deep Learning with Python

Training Loop

Step 1 is easy: just some I/O code

Steps 2 & 3 just consist of a a handful of tensor operations, also easy

Step 4, updating the network's weights, is difficult

Given an individual weight coefficient in the network, how can we compute whether the coefficient should be increased or deceased, and by how much?

Slide credit: F. Chollet, Deep Learning with Python

Naive Strategy for Updating Weights

Freeze all weights in the network except the one scalar coefficient being considered, and try different values for it. Repeat for all coefficients in the network.

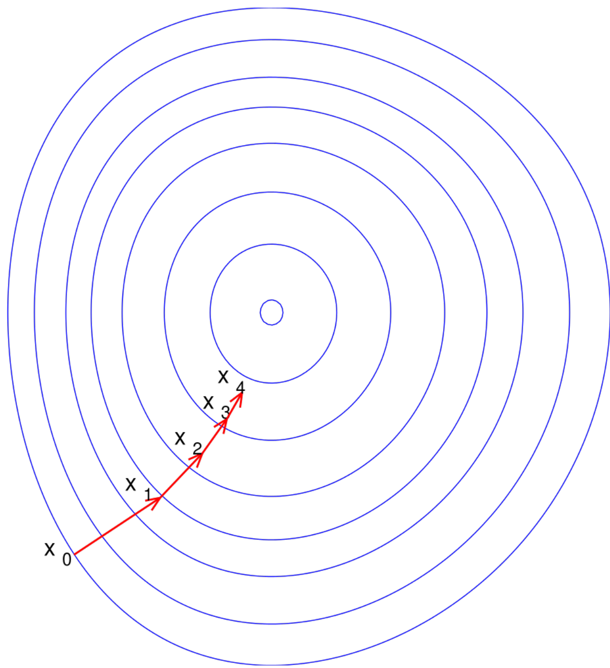
Example:

Initial value of coefficient: 0.3	Corresponding loss of net: 0.5
New value of coefficient: 0.35	Corresponding loss of net: 0.6
New value of coefficient: 0.25	Corresponding loss of net: 0.4

Why is this algorithm bad?

Slide credit: F. Chollet, Deep Learning with Python

Gradient-Based Learning



Take advantage of the fact that all operations used in the network are **differentiable**, and compute the **gradient** of the loss with respect to the network's coefficients.

Move coefficients in the opposite direction from the gradient, thus decreasing the loss

Slide credit: F. Chollet, Deep Learning with Python

Derivatives

Consider a continuous, smooth function $f(x) = y$, mapping a real number x to a new real number y

The function is continuous: a small change in x can only result in a small change in y

If x is increased by a small factor ε_x this results in a small ε_y change to y:

$$f(x + \varepsilon_x) = y + \varepsilon_y$$

Slide credit: F. Chollet, Deep Learning with Python

Derivatives

$f(x) = y$, is a smooth function (the curve doesn't have any abrupt angles)

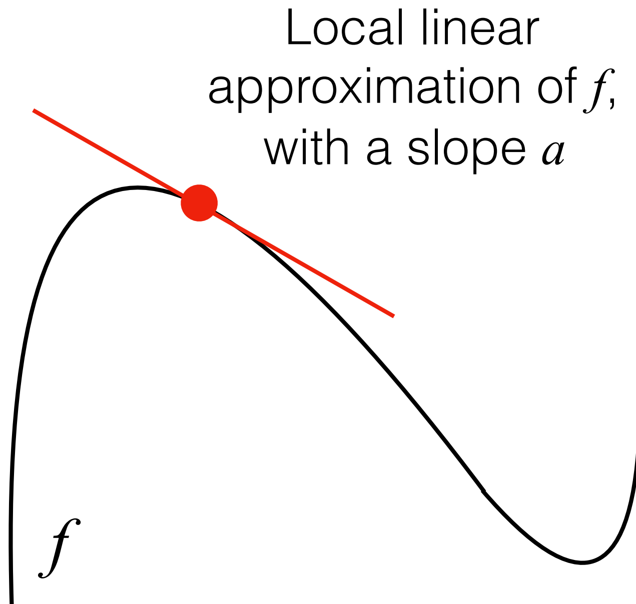
When ε_x is small enough, around a certain point p, it's possible to approximate f as a linear function of slope a, so that ε_y becomes a * ε_x:

$$f(x + \varepsilon_x) = y + a \times \varepsilon_y$$

This linear approximation is valid only when x is close enough to p

Slide credit: F. Chollet, Deep Learning with Python

Derivative of f in p



The slope a is called the derivative of f in p

If a is negative, it means a small change of x around p will result in a decrease of $f(x)$

If a is positive, a small change in x will result in an increase of $f(x)$

Absolute value of a tells you how quick this increase or decrease will happen

Slide credit: F. Chollet, Deep Learning with Python

Differentiable functions

For every differentiable function $f(x)$, there exists a derivative function $f'(x)$ that maps values of x to the slope of the local linear approximation of f in those points

Examples:

The derivative of $\cos(x)$ is $-\sin(x)$

The derivative of $f(x) = a \times x$ is $f'(x) = a$

The derivative completely describes how $f(x)$ evolves as you change x

If you want to reduce the value of $f(x)$, you just need to move x a little in the opposite direction of the derivative

Slide credit: F. Chollet, Deep Learning with Python

Derivative of a tensor operation: the gradient

A **gradient** is the generalization of the concept of derivatives to functions of multidimensional inputs

Consider an input vector x, a matrix W, a target y, and a loss function loss. You can use W to compute a target candidate y_{pred} and compute the loss, or mismatch, between the target candidate y_{pred} and the target y:

$$y_{\mathrm{pred}} = \mathrm{dot}(W, x)$$

$$\mathrm{loss_value} = \mathrm{loss}(y_{\mathrm{pred}}, y)$$

If data inputs x and y are frozen, then:

$$\mathrm{loss_value} = f(W)$$

Slide credit: F. Chollet, Deep Learning with Python

Derivative of a tensor operation: the gradient

Let's say the current value of W is $W0$

The derivative of f in $W0$ is a tensor $\text{gradient}(f)(W0)$ with the same shape as W,

- Each coefficient $\text{gradient}(f)(W0)[i, j]$ indicates the direction and magnitude of the change in loss observed when modifying $W0[i, j]$
- The tensor gradient $\text{gradient}(f)(W0)$ is the gradient of $f(W) = \text{loss_value}$ in $W0$

$\text{gradient}(f)(W0)$ can be interpreted as the tensor describing the curvature of $f(W)$ around $W0$

Slide credit: F. Chollet, Deep Learning with Python

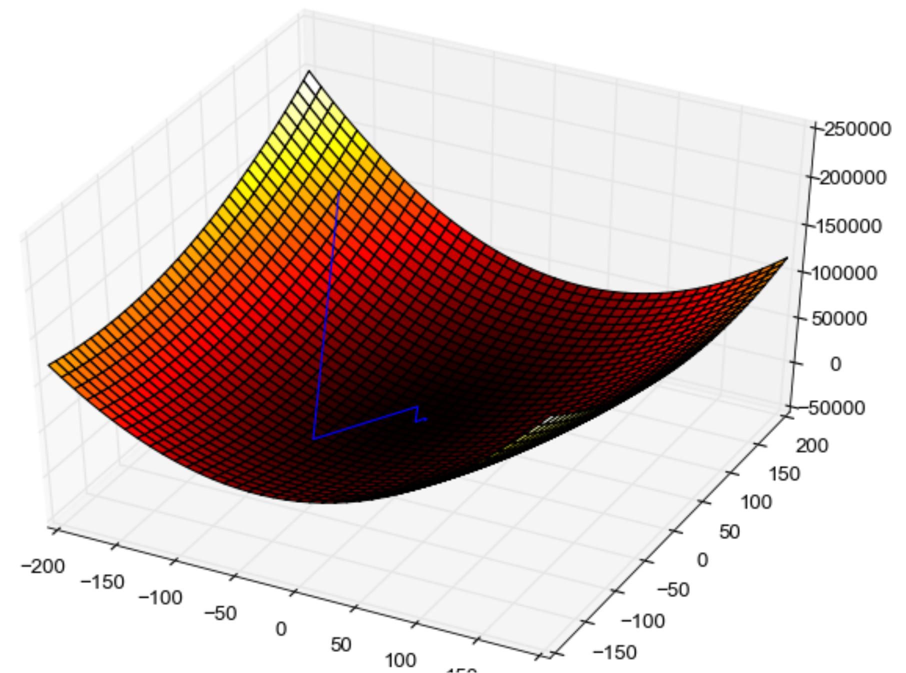
Derivative of a tensor operation: the gradient

You can reduce $f(W)$ by moving W in the opposite direction from the gradient

Example:

$$W1 = W0 - \text{step} \times \text{gradient}(f)(W0)$$

Moves go against the curvature, which intuitively should put you lower on the curve



Gradient descent example © BY-SA 3.0 Роман Сузи

Slide credit: F. Chollet, Deep Learning with Python

Analytical Solution?

Yes: solve the equation $\mathbf{gradient}(f)(W) = 0$ for W.

This is a polynomial equation of N variables, where N is the number of coefficients in the network

How many coefficients are we typically dealing with in a modern neural network?

Finding the best weights: Hill Descent

How do we get to the bottom of the deepest valley?

How do we do this if we don't have gravity?

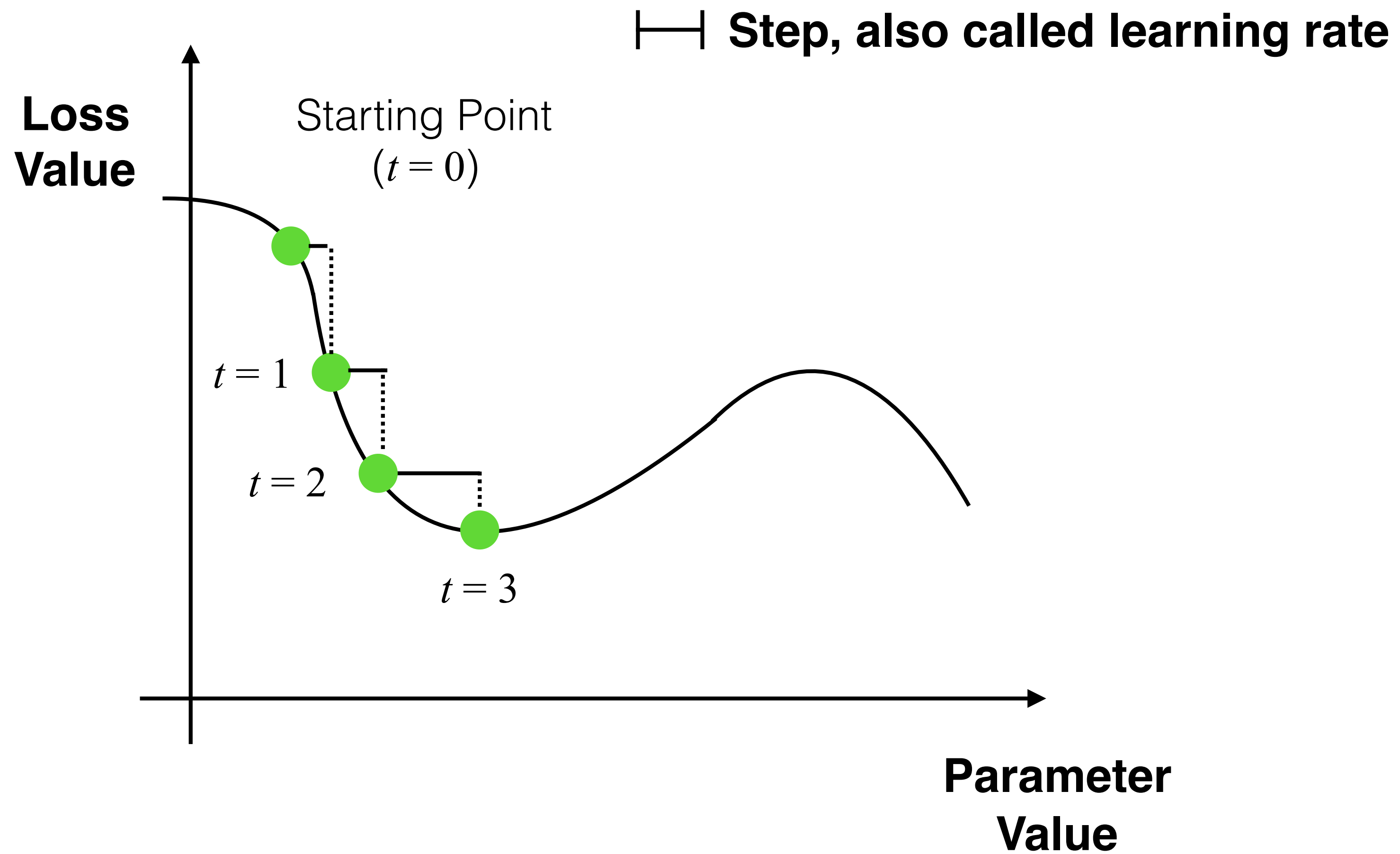
Image credit: Abu-Mostafa, Magdon-Ismail, Lin, Learning from Data

Mini-batch Stochastic Gradient Descent (SGD)

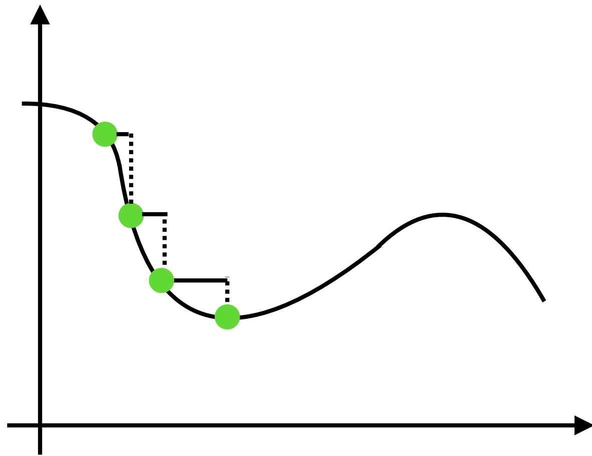
1. Draw a batch of training samples x and corresponding targets y
2. Run the network on x to obtain predictions **y_pred**
3. Compute the loss of the network on the batch a measure of mismatch between **y_pred** and **y**
4. Compute the gradient of the loss with respect to the network's parameters (a *backward* pass)
5. Move the parameters a little in the opposite direction from the gradient, thus reducing the loss on the batch a bit

Slide credit: F. Chollet, Deep Learning with Python

SGD down a 1D loss curve



Selecting a step value



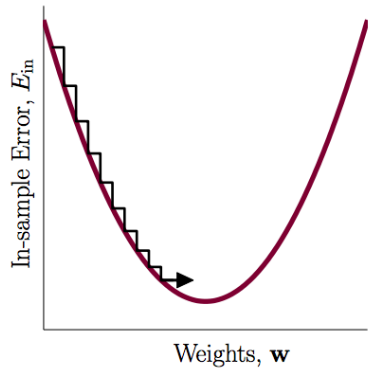
It's important to pick a reasonable value for the step factor

What happens if we choose a value that is too small?

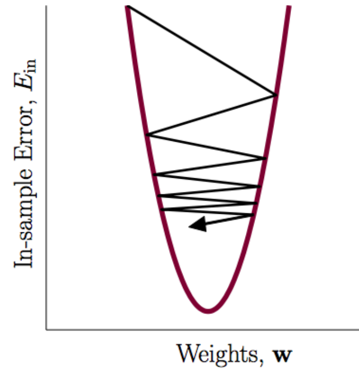
What happens if we choose a value that is too large?

Slide credit: F. Chollet, Deep Learning with Python

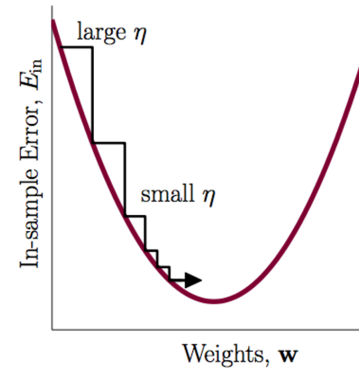
Too Small

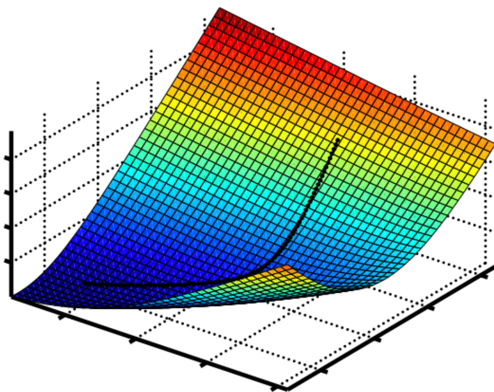


Too Large

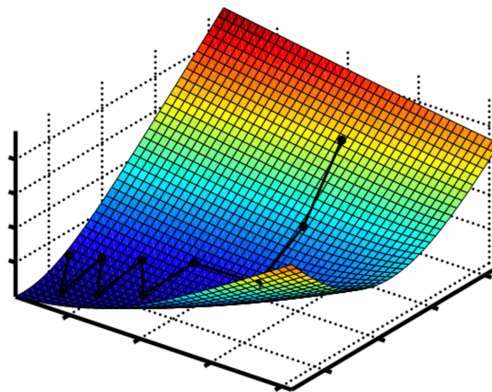


Variable (Just Right)

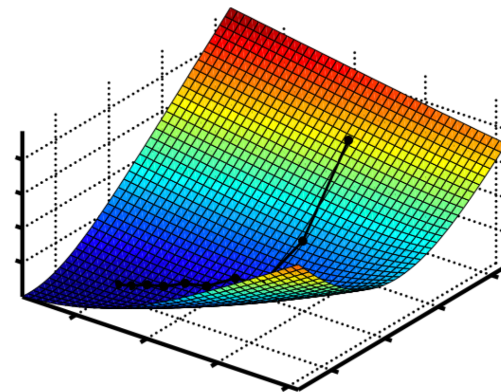




step = 0.1; 75 steps



step = 2; 10 steps



step = variable; 10 steps

Image credit: Abu-Mostafa, Magdon-Ismail, Lin, Learning from Data

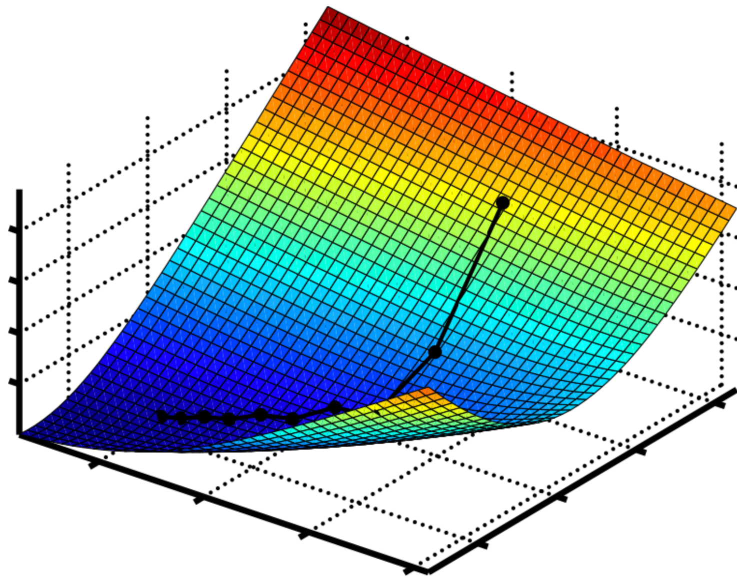
Other SGD Variants

True SGD: draw a single sample and target at each iteration, rather than drawing a batch of data

Batch SGD: run every step on *all* available data

- Each update would be more accurate, but far more expensive

Mini-Batch SGD is an efficient compromise between the two strategies



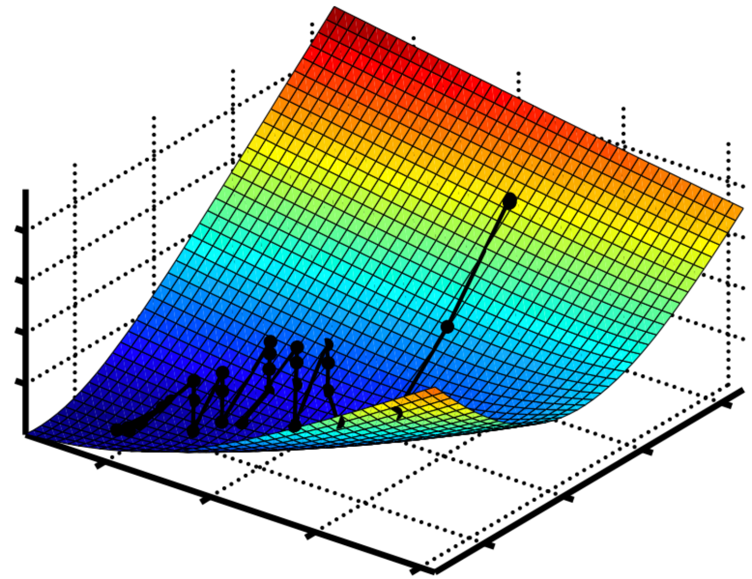
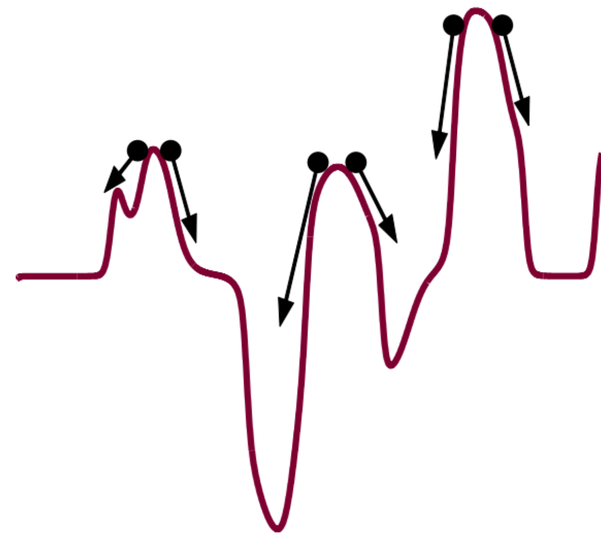


Image credit: Abu-Mostafa, Magdon-Ismail, Lin, Learning from Data

Momentum

How can we mitigating the problem off getting stuck in bad local minima?

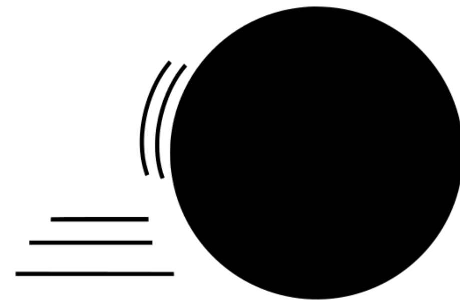


A **momentum** term addresses two problems with SGD: local minima and convergence speed

Image credit: Abu-Mostafa, Magdon-Ismail, Lin, Learning from Data

Momentum

Imagine the optimization as a small ball rolling down the loss curve. If it has enough momentum, it won't get stuck in a ravine and will end up at the global minimum.



Implement by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (from past acceleration).

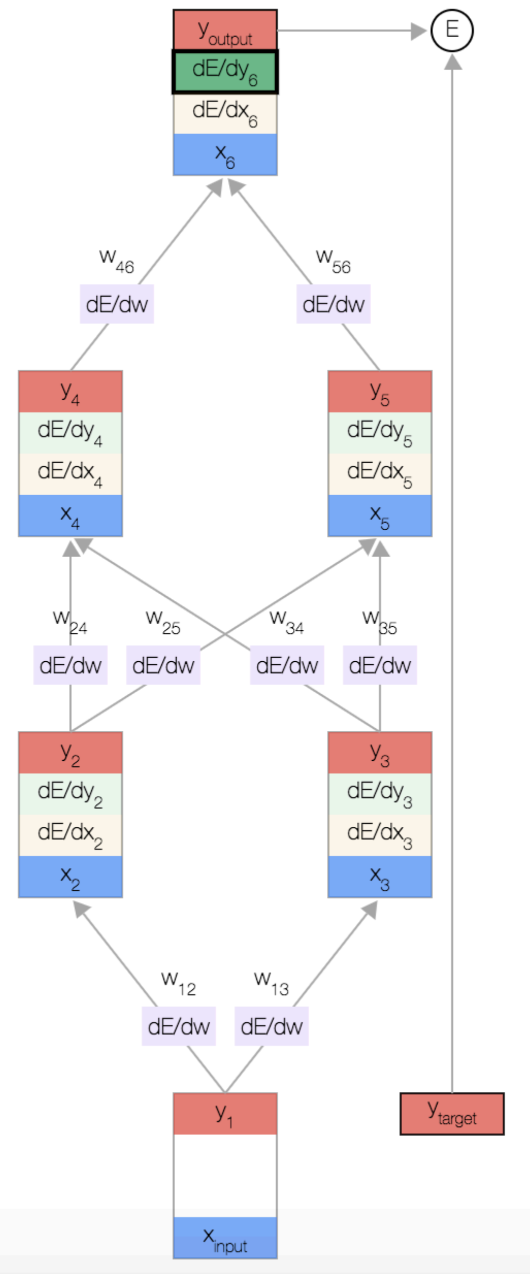
i.e., update parameter w based on the current gradient and on the previous parameter update

How do we compute the gradient?

Back propagation

Let's begin backpropagating the error derivatives. Since we have the predicted output of this particular input example, we can compute how the error changes with that output. Given our error function $E = \frac{1}{2}(y_{output} - y_{target})^2$ we have:

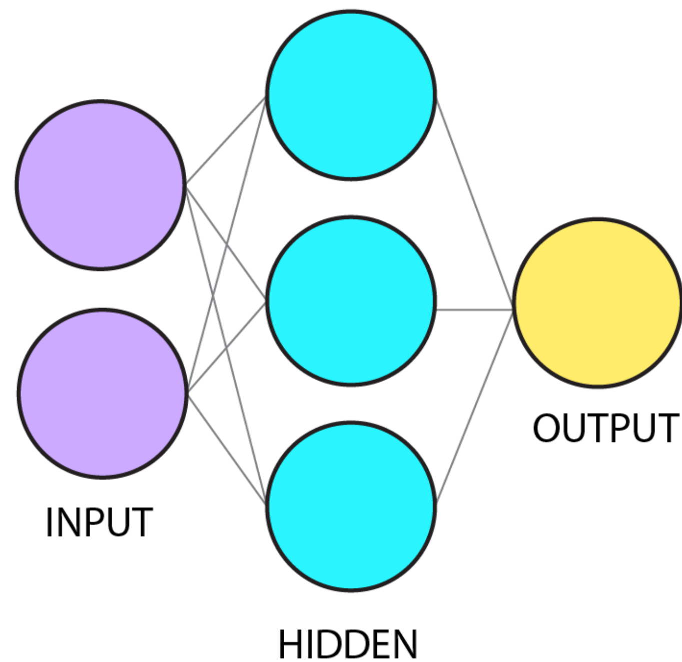
$$\frac{\partial E}{\partial y_{output}} = y_{output} - y_{target}$$



https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/

PyTorch Example

How does PyTorch train the following network?



Example adapted from: http://bit.ly/2Er1Cy0

What is the learning rule used by the brain?