First-Order Optimization Techniques

Instructor: Hui Guan

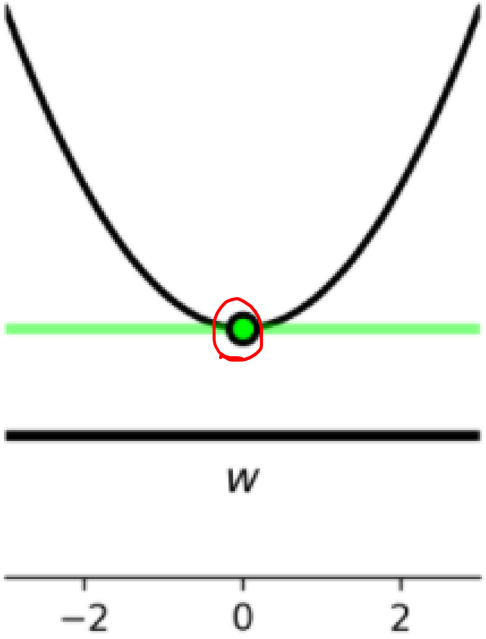
Slides adapted from: https://github.com/jermwatt/machine_learning_refined

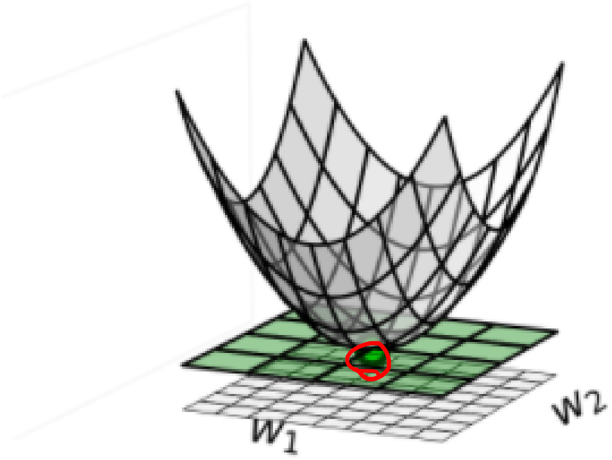
Outline

- First order optimality condition
- Coordinate descent
- Compute gradients efficiently
- Gradient descent
- Issues with gradient descent

The first-order condition

- Below we plot a quadratic functions in two and three dimensions, and mark the global minimum point on each with a green point.
- The tangent line/hyperplane is perfectly flat, indicating that the first derivative(s) is exactly zero at the function's minimum.

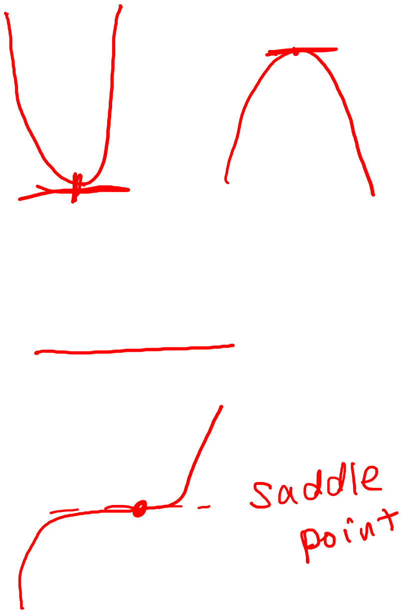




- Codifying this in the language of mathematics, when ***N=1*** any point ***v*** where

$$\frac{\mathrm{d}}{\mathrm{d}w} g\left(v\right) = 0$$

is a potential minimum.



- Analogously with general ***N*** dimensional input, any ***N*** dimensional point $\mathbf{v}$ where *every* **partial derivative** of ***g*** is zero, that is

$$
\begin{gathered}
\frac{\partial}{\partial w_{\underline{1}}} g(\mathbf{v}) = 0 \\
\frac{\partial}{\partial w_2} g(\mathbf{v}) = 0 \\
\vdots \\
\frac{\partial}{\partial w_{\underline{N}}} g(\mathbf{v}) = 0
\end{gathered}
$$

is a potential minimum.

- This system of ***N*** equations is naturally referred to as the *first order system of equations (first-order optimality condition)*.
- We can write the first order system more compactly using gradient notation as

$$\boxed{\nabla g\left(\mathbf{v}\right) = \underline{\vec{\mathbf{0}}}_{N\times 1}.}$$

- However two problems with the first order characterization of minima.
- First off, with few exceptions, **it is virtually impossible to solve a general function's first order systems of equations 'by hand'.**
- The other problem: the *first order optimality condition* does not define only minima of a function, but other points as well.
    - The first order condition also equally characterizes *maxima* and *saddle points* of a function - as we see in a few simple examples below.

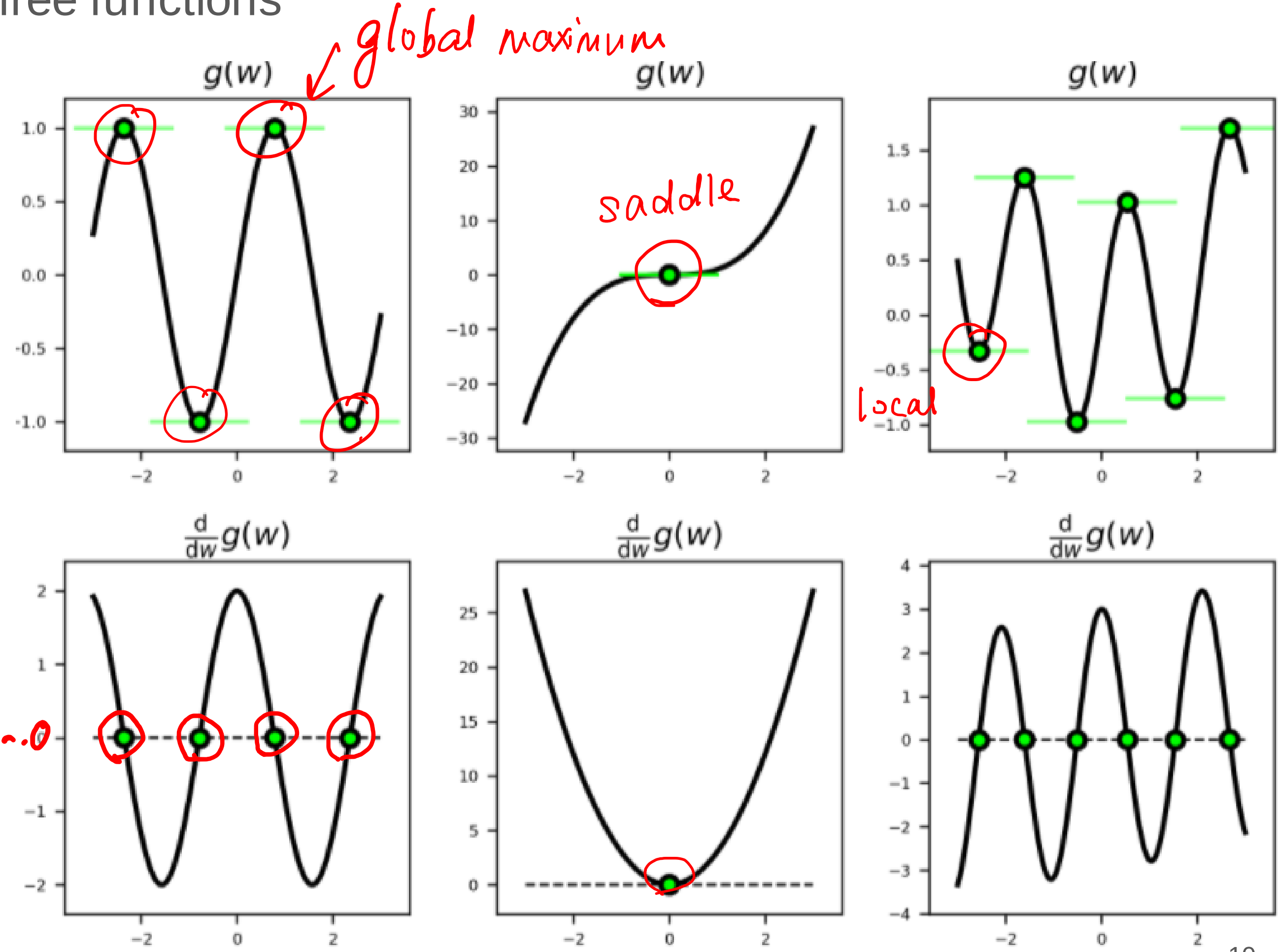
Example: Finding points of zero derivative for single-input functions graphically

- Below we plot the three functions

$$g(w) = \sin(2w)$$

$$g(w) = w^3$$

$$g(w) = \sin(3w) + 0.1w^2$$



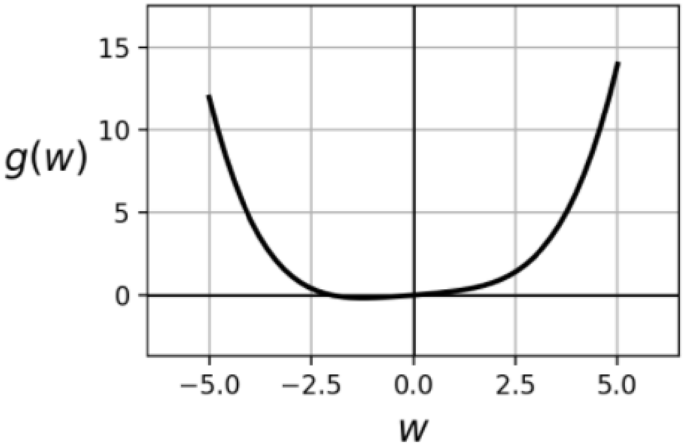
- *local minima* or points that are the smallest with respect to their immediate neighbors, like the one around the input value ***w=2*** in the right panel
- *local and global maxima* or points that are the largest with respect to their immediate neighbors, like the one around the input value ***w=-2*** in the right panel
- *saddle points* like the one shown in the middle panel, that are neither maximal nor minimal with respect to their immediate neighbors
- *Taken together all such points are collectively referred to as stationary points or critical points.*

Example: A simple looking function with difficult to compute (algebraically) global minimum

- Solving the first order equation for even a simple looking function can be quite challenging.
- Take, for example, the simple degree four polynomial

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right) \qquad \frac{\partial g(w)}{\partial w} = \frac{1}{50}\left(4w^3 + 2w + 10\right) = 0$$

- This is plotted over a short range of inputs containing its global minimum below.



- The first order system here can be easily computed as

  $$\frac{\mathrm{d}}{\mathrm{d}w} g(w) = \frac{1}{50}\left(4w^3 + 2w + 10\right) = 0$$

  This simplifies to

  $$2w^3 + w + 5 = 0$$

- This has three possible solutions, but the one providing the minimum of the function $g(w)$ is

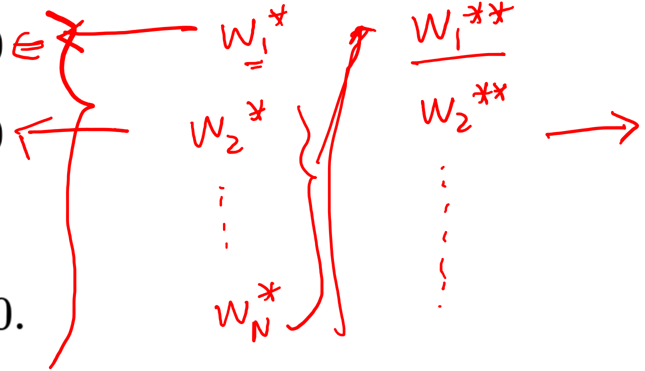
$$w = \frac{\sqrt[3]{\sqrt{2031}-45}}{6^{\frac{2}{3}}} - \frac{1}{\sqrt[3]{6\left(\sqrt{2031}-45\right)}}$$

which can be computed - after much toil - using centuries old tricks developed for just such problems.

Coordinate descent

- If we write out the first order system one equation at-a-time we have

$$
\begin{aligned}
\frac{\partial}{\partial w_1} g(\mathbf{v}) &= 0 \\
\frac{\partial}{\partial w_2} g(\mathbf{v}) &= 0 \\
&\vdots \\
\frac{\partial}{\partial w_N} g(\mathbf{v}) &= 0.
\end{aligned}
$$



- While this system cannot often be solved in closed form, a simple idea does lead to numerical approach to approximating it instances where ***each individual equation* can be easily solved**.

- The idea is this**: instead of trying to solve the system of equations *at once* we solve each partial derivative equation *one at-a-time***.
- This is often called *coordinate descent*, since in solving each we move along the coordinate axes coordinate-wise (one at-a-time).

- To perform this coordinate descent we initialize at a point $\mathbf{w}^0$, updating its first coordinate by solving

$$\frac{\partial}{\partial w_1} g\left(\mathbf{w}^0\right) = 0$$

for the optimal first weight $w_1^\star$.

- We do this again, and again, for each coordinate.

- Continuing this pattern to update the n^{th} weight we solve

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{n-1}\right) = 0$$

for $w_n^{\star}$, and update the n^{th} weight using this value forming the updated set of weights $\mathbf{w}^{n}$.

- After we sweep through all ***N*** weights a single time we can refine our solution by sweeping through the weights again.
- At the k^{th} such sweep we update the n^{th} weight by solving the single equation

$$\frac{\partial}{\partial w_n} g\left(\mathbf{w}^{N(k-1)+n-1}\right) = 0$$

and update the n^{th} weight of $\mathbf{w}^{k+n-1}$, and so on.

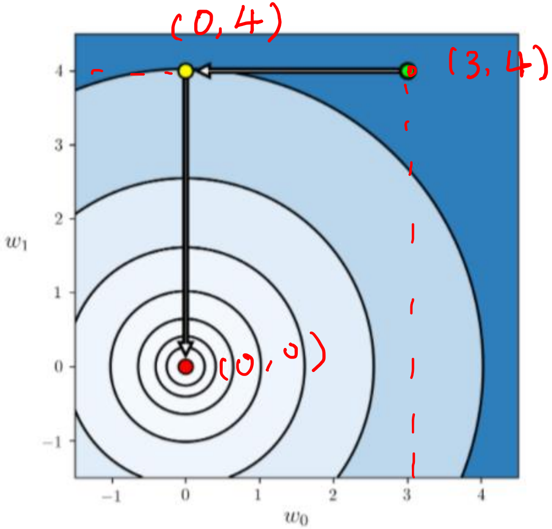
Example: Minimizing convex quadratic functions via first order coordinate descent

- First we use this algorithm to minimize the simple quadratic

$$g(w_0, w_1) = w_0^2 + w_1^2 + 2$$

$$\begin{cases} \frac{\partial g}{\partial w_0} = 2w_0 \\ \frac{\partial g}{\partial w_1} = 2w_1 \end{cases}$$

- We initialize at $\mathbf{w} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$ and run ***1*** iteration of the algorithm - that is all it takes to perfectly minimize the function, as shown below.



$$\frac{\partial g}{\partial w_0} = 2w_0 = 0 \rightarrow \underline{w_0 = 0}$$

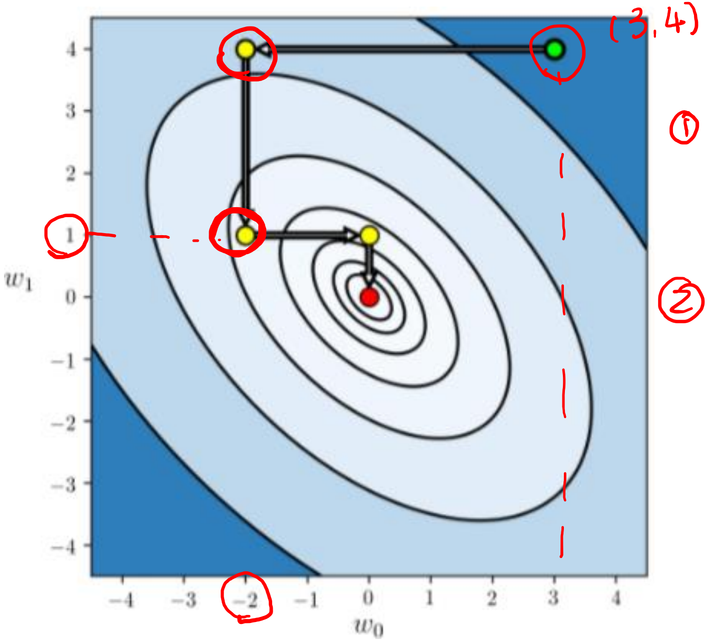
$$\frac{\partial g}{\partial w_1} = 2w_1 = 0 \Rightarrow w_1 = 0$$

- Below we show a run of ***2*** iterations of the method at the same initial point for the quadratic

$$\frac{\partial g}{\partial w_0} = 4w_0 + 2w_1$$

$$\frac{\partial g}{\partial w_1} = 4w_1 + 2w_0$$

$$g(w_0, w_1) = 2w_0^2 + 2w_1^2 + 2w_0 w_1 + 20$$



① $\frac{\partial g}{\partial w_0} = 4w_0 + 2w_1 = 0 \quad \Big| \quad w_1 = 4$

$w_0 = -\frac{1}{2} w_1 = -2$

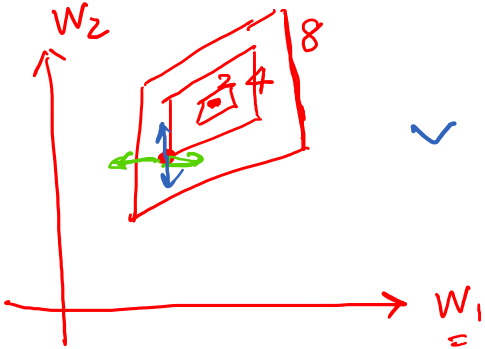
② $\frac{\partial g}{\partial w_1} = 4w_1 + 2w_0 = 0 \quad \Big| \quad w_0 = -2$

$w_1 = +1$

True/False on Coordinate Descent

- [[✓ True]/False] Coordinate descent is applicable in both differentiable and derivative-free contexts.

- [[✓ True]/False] Coordinate descent assumes that minimizing a multi-variable function can be achieved by minimizing along one direction at a time (e.g., solving a univariate optimization problem in a loop)



Computing Gradients Efficiently

- You likely learned a bunch of basic rules for computing derivatives in school, and can compute simple examples like $g(w) = w^3$ and $g(w) = \sin(w)$
- But what about this one?

$$g\left(w_1, w_2\right) = 2^{\sin\left(0.1w_1^2 + 0.5w_2^2\right)} \tanh\left(w_2^4 \tanh\left(w_1 + \sin\left(0.2w_2^2\right)\right)\right)$$

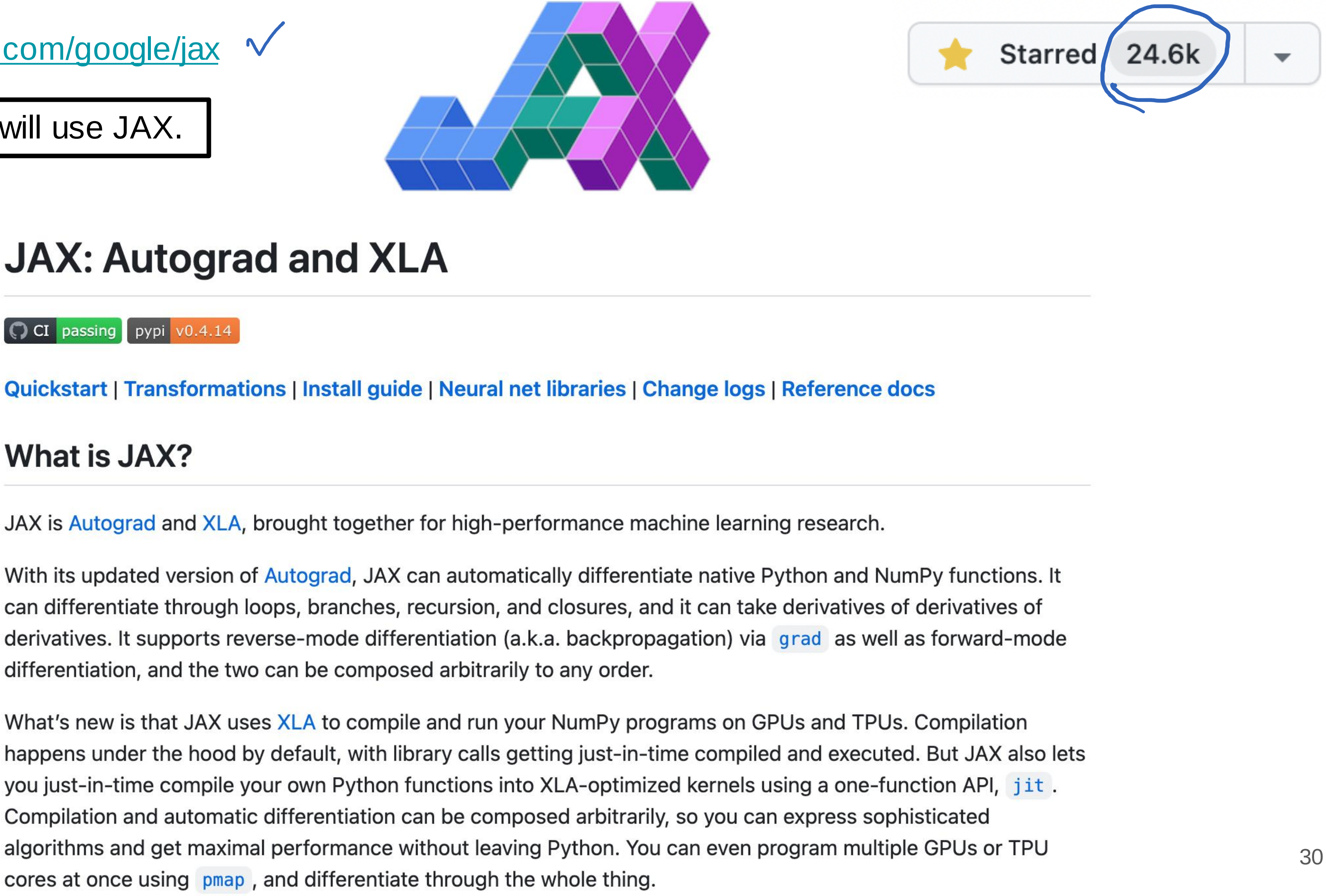
- You *could* compute the derivatives yourself, since the process is simple and repetitive, but its also *boring* and *time cosuming* and you could easily mess up.
- Your time is better spent doing more thought-intensive things, so why not use a calculator instead?

- A gradient calculator or *Automatic Differentiator* allows *you* to compute with much greater effeciency and accuracy, and empowers you to use the fruits of gradient computation for more important tasks.

- A powerful and easy to use \`Python\` Automatic Differentiator called \`autograd\` and more recently updated to “**Jax**”.

https://github.com/google/jax ✓

Your 2nd HW will use JAX.



JAX: Autograd and XLA

CI passing pypi v0.4.14

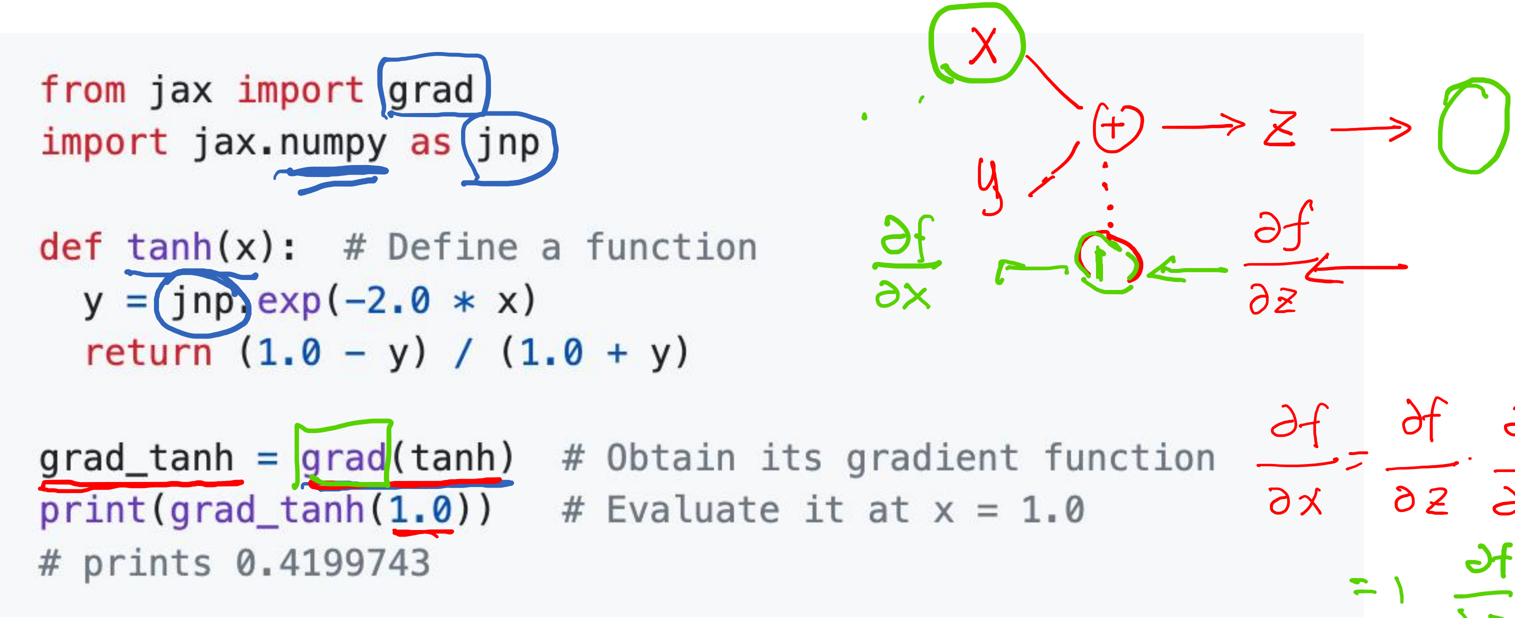
Quickstart | Transformations | Install guide | Neural net libraries | Change logs | Reference docs

What is JAX?

JAX is Autograd and XLA, brought together for high-performance machine learning research.

With its updated version of Autograd, JAX can automatically differentiate native Python and NumPy functions. It can differentiate through loops, branches, recursion, and closures, and it can take derivatives of derivatives of derivatives. It supports reverse-mode differentiation (a.k.a. backpropagation) via `grad` as well as forward-mode differentiation, and the two can be composed arbitrarily to any order.

What's new is that JAX uses XLA to compile and run your NumPy programs on GPUs and TPUs. Compilation happens under the hood by default, with library calls getting just-in-time compiled and executed. But JAX also lets you just-in-time compile your own Python functions into XLA-optimized kernels using a one-function API, `jit`. Compilation and automatic differentiation can be composed arbitrarily, so you can express sophisticated algorithms and get maximal performance without leaving Python. You can even program multiple GPUs or TPU cores at once using `pmap`, and differentiate through the whole thing.



```
from jax import grad
import jax.numpy as jnp

def tanh(x):  # Define a function
  y = jnp.exp(-2.0 * x)
  return (1.0 - y) / (1.0 + y)

grad_tanh = grad(tanh)  # Obtain its gradient function
print(grad_tanh(1.0))   # Evaluate it at x = 1.0
# prints 0.4199743
```

x
+
z
y
1
$\frac{\partial f}{\partial x}$
$\frac{\partial f}{\partial z}$

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial z} \cdot \frac{\partial z}{\partial x}$$

$$= 1 \; \frac{\partial f}{\partial z}$$

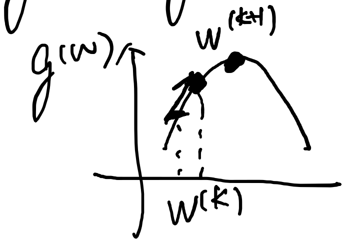
Note: JAX cannot differentiate numpy functions, but it can differentiate jax.numpy functions.

Gradient Descent

$$\boxed{\begin{aligned} d^k &= \nabla g(w^{k-1}) \\ w^k &= w^{k-1} + \alpha \cdot d^k \end{aligned}}$$

gradient ascent

$w^* = \arg\max g(w)$



- Remember, a general local optimization method looks like

$$\mathbf{w}^{k} = \mathbf{w}^{k-1} + \boxed{\alpha \mathbf{d}^{k}}.$$

- Here $\mathbf{d}^k$ are *descent direction* vectors and α is called the *steplength parameter*. The basic form of gradient descent is:

random local search

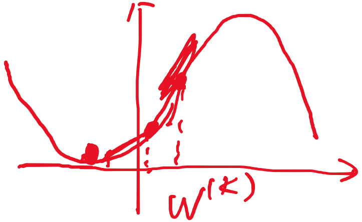
$\vec{d} \sim \text{sample}(\quad)$

s.t. $\underline{\|\vec{d}\|_2 = 1}$

$$\boxed{\mathbf{d}^{k} = \underline{-\nabla g\left(\mathbf{w}^{k-1}\right)}}$$

$$\mathbf{w}^{k} = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$$

$w^* = \arg\min g(w)$



- Q1: How do we set the α parameter in general?

$$\mathbf{w}^{k} = \mathbf{w}^{k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$$

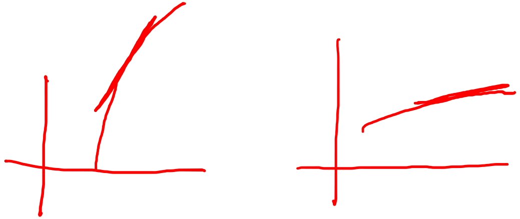
- Q1: How do we set the α parameter in general?
- Popular approaches are precisely those introduced in the (comparatively simpler) context of zero order methods: that is **fixed** and **diminishing steplength** choices.

- Q2: When does gradient descent stop?

$$\mathbf{w}^{\,k} = \mathbf{w}^{\,k-1} - \alpha \nabla g\left(\mathbf{w}^{k-1}\right)$$

- Q2: When does gradient descent stop?
- Technically (when α is chosen well) the algorithm will ***halt near stationary points of a function, typically minima or saddle points***.

- Below we provide the generic pseudo-code of the gradient descent algorithm which will be used in a variety of examples.



generated random

The gradient descent algorithm

```
input: function $g$, steplength $\alpha$, maximum number of steps $K$, and initial point $\mathbf{w}^0$
for $k = 1...K$
        $\mathbf{w}^k = \mathbf{w}^{k-1} - \alpha\nabla g\left(\mathbf{w}^{k-1}\right)$
output: history of weights $\left\{\mathbf{w}^k\right\}_{k=0}^K$ and corresponding function evaluations $\left\{g\left(\mathbf{w}^k\right)\right\}_{k=0}^K$
```

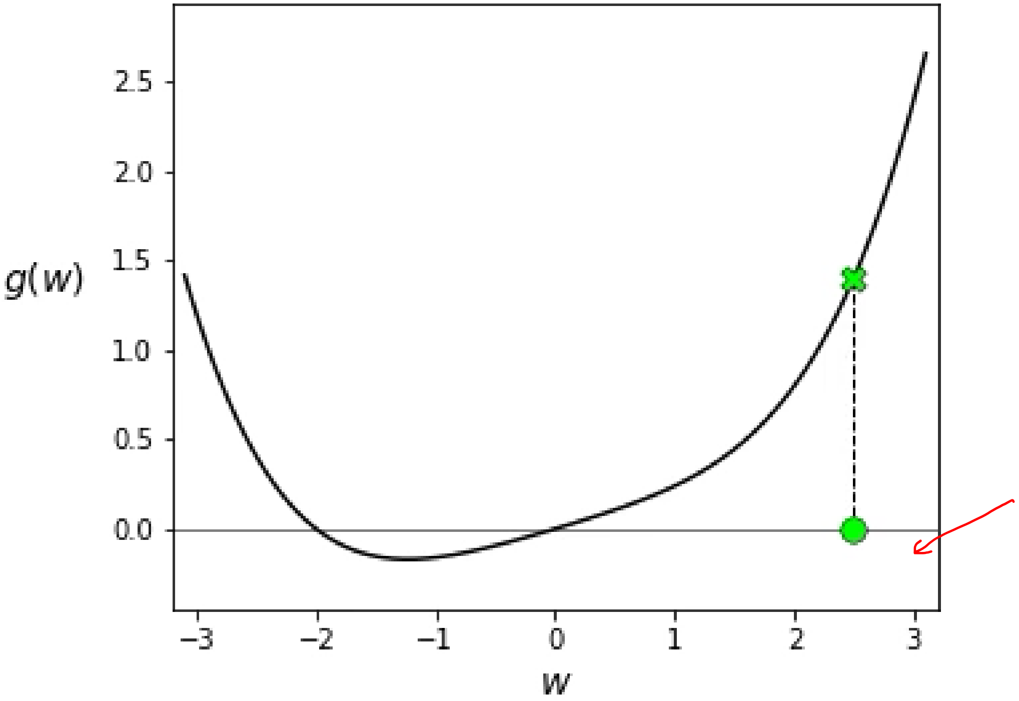
cost

Example: A convex single input example

- Below we animate the use of gradient descent to minimize the polynomial function

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right).$$

- Here $w_0 = 2.5$ and $\alpha = 1$

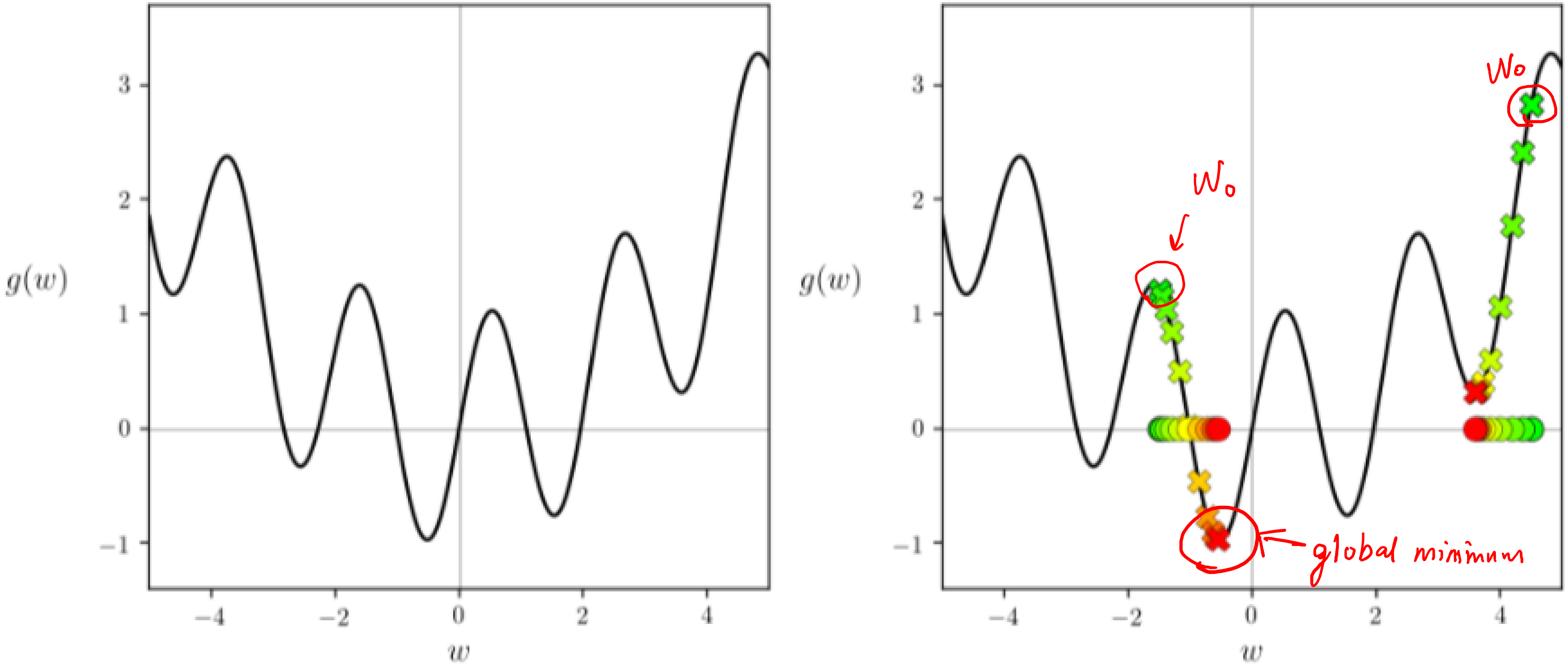

g(w)
2.5
2.0
1.5
1.0
0.5
0.0
−3
−2
−1
0
1
2
3
w

Example: A non-convex single input example

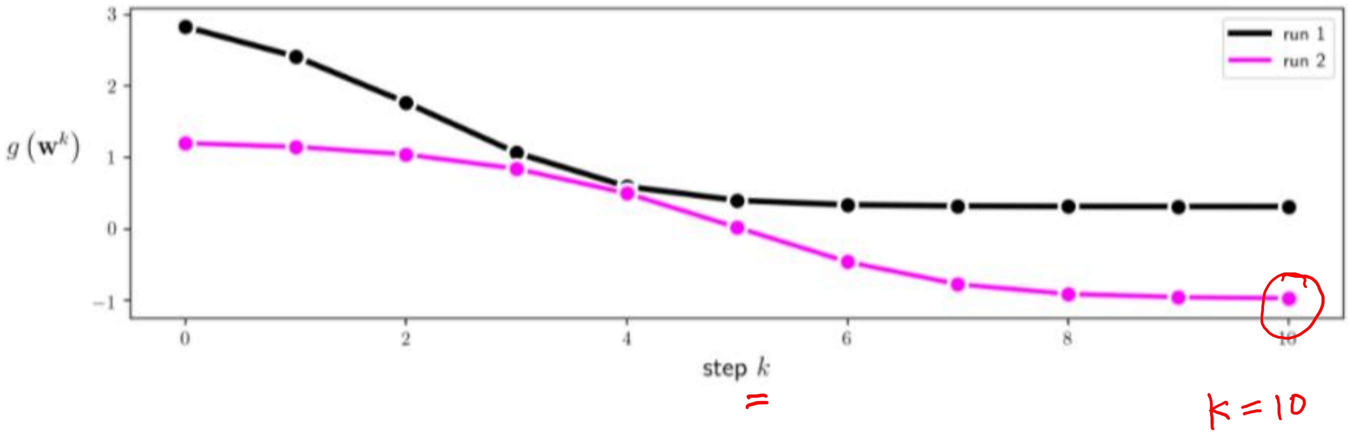
- Now we show the result of running gradient descent several times to minimize the function

$$g(w) = \sin(3w) + 0.1w^2$$

- For general non-convex functions like this one, several runs (of any local optimization method) can be necessary to determine points near global minima.

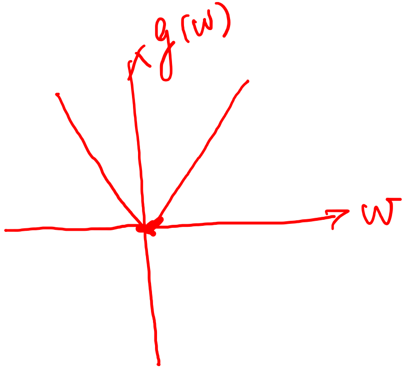

g(w)
w
3
2
1
0
−1
−4
−2
0
2
4
g(w)
w
3
2
1
0
−1
−4
−2
0
2
4
W₀
W₀
global minimum

- Viewing the *cost function history plot* allows us to view the progress of gradient descent, regardless of the function's input dimension.
- these plots are a valuable debugging tool, as well as a valuable tool for selecting proper values for the steplength



True. False

- (true？False?) Since the magnitude of gradients is diminishing ✗ when it is close to the minimum, diminishing learning rate is not necessary.



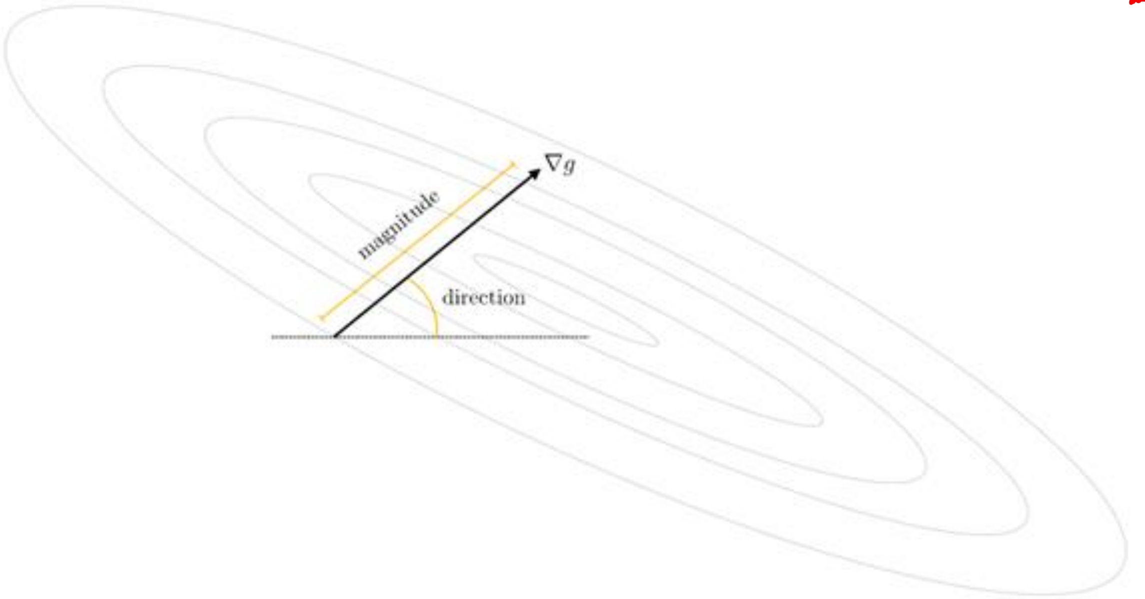
$$w^k = w^{k-1} - \alpha \cdot \underbrace{\nabla g(w^{k-1})}$$

$$g(w) = |w|$$

Two issues with the negative gradient as a descent direction

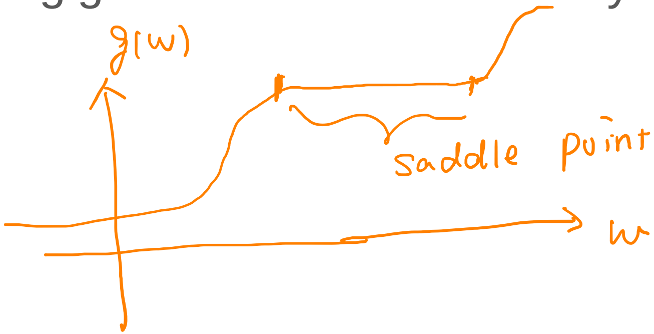
- Like any *vector* the negative gradient always consists fundamentally of a *direction* and a *magnitude*.

$$\vec{W}^{k} = \vec{W}^{k-1} - \alpha \cdot \boxed{\nabla g(w^{k-1})}$$



- Issue 1: The *direction* of the negative gradient can *rapidly oscillate* or *zig-zag* during a run of gradient descent, often producing *zig-zagging* steps that take considerable time to reach a near mininum point.

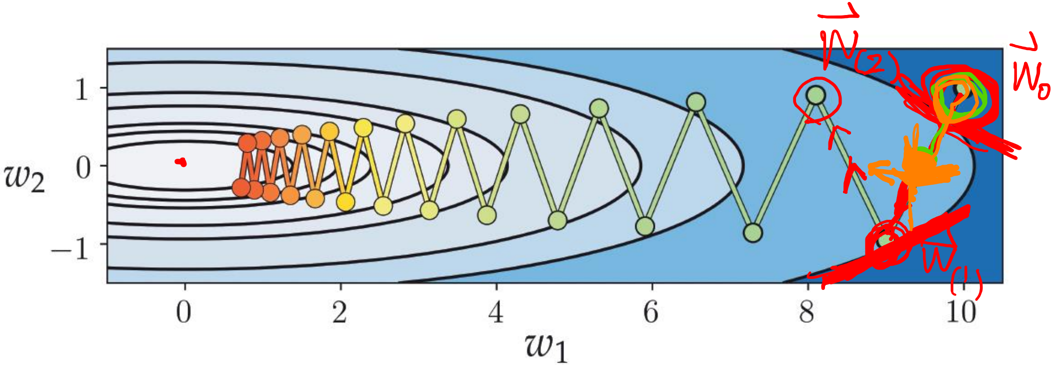
- Issue 2: The *magnitude* of the negative gradient can *vanish rapidly* near stationary points, leading gradient descent to slowly crawl near minima and saddle points.



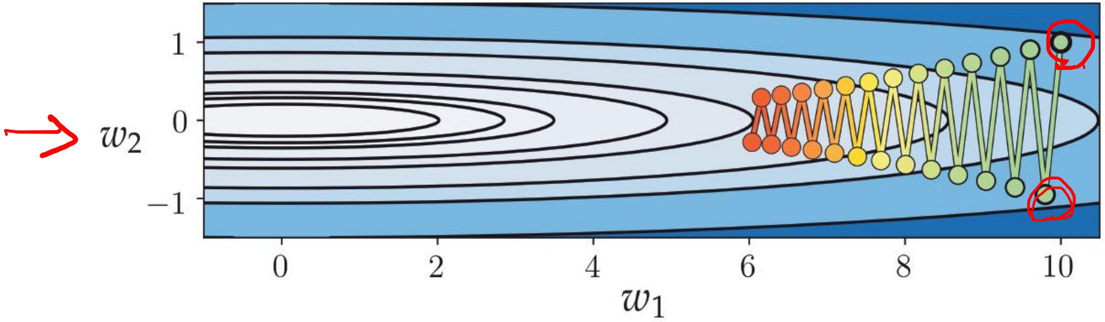
Example: Zig-zagging behavior of gradient descent on three simple quadratic functions

$\vec{w} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}$ $\vec{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$ $C \in \mathbb{R}^{2\times 2}$

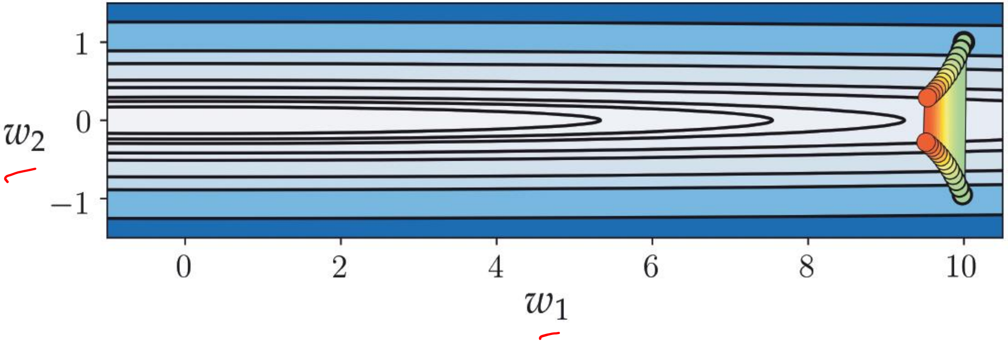
- We illustrate the zig-zag behavior of gradient descent with three $N = 2$ dimensional quadratic $g(\underline{\mathbf{w}}) = \underline{a} + \underline{\mathbf{b}}^T\mathbf{w} + \mathbf{w}^T\mathbf{C}\mathbf{w}$ $\Rightarrow c_{11}w_1^2 + c_{12}w_1w_2 + \cdots w_2^2 + c_{21}w_1w_2$ $+$
- Not much progress is made with the third quadratic at all due to the large amount of zig-zagging.
- We can also see the cause of this zig-zagging: the negative gradient direction constantly points perpindicular to the contours of the function (this can be especially seen in the third case).



$$C = \begin{bmatrix} 0.5 & 0 \\ 0 & 12 \end{bmatrix}$$



$$C = \begin{bmatrix} 0.1 & 0 \\ 0 & 12 \end{bmatrix}$$



$$C = \begin{bmatrix} 0.01 & 0 \\ 0 & 12 \end{bmatrix}$$

- It is the true that we can ameilorate this zig-zagging behavior by *reducing the steplength value*, as shown below.
- However this does not solve the underlying problem that zig-zagging produces - which is slow convergence.
- Typically in order to ameliorate or even eliminate zig-zagging this way requires a very small steplength, which leads back to the fundamental problem of slow convergence.

E[f]

Momentum-accelerated GD

β ∈ [0.7, 1)

moving average

$$\mathbf{d}^{k-1} = \beta\,\mathbf{d}^{k-2} + (1-\beta)\left(-\nabla g\left(\mathbf{w}^{k-1}\right)\right)$$

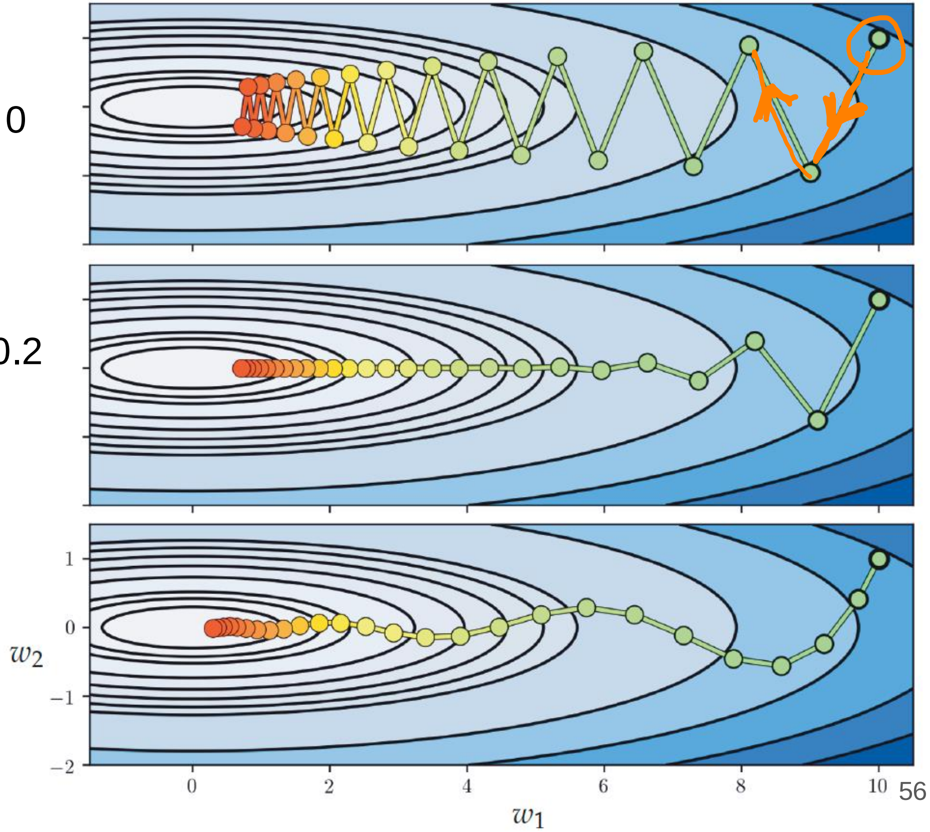
$$\mathbf{w}^{k} = \mathbf{w}^{k-1} + \alpha\,\mathbf{d}^{k-1}.$$

[0.001 , 0.01]

Beta = 0

Beta = 0.2

Beta = 0.7



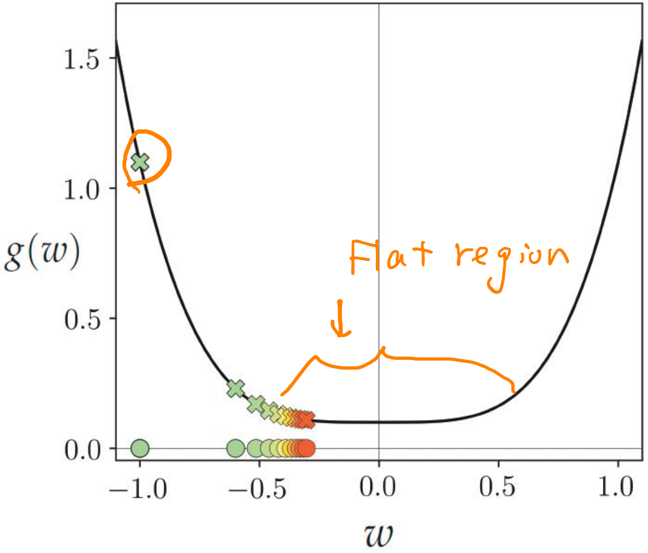
Example: Slow-crawling behavior of gradient descent near the minimum of a function

- Below we show another example run of gradient descent using a function

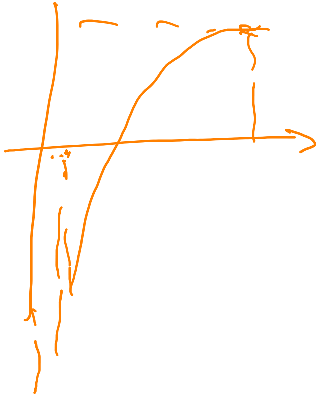
$$g(w) = w^4 + 0.1$$

whose minimum is at the origin.

- The steps get very small and crawls as we get closer and closer to the minimum of this function.



$\alpha \sim \log(|\nabla g(w)|)$



Normalized GD

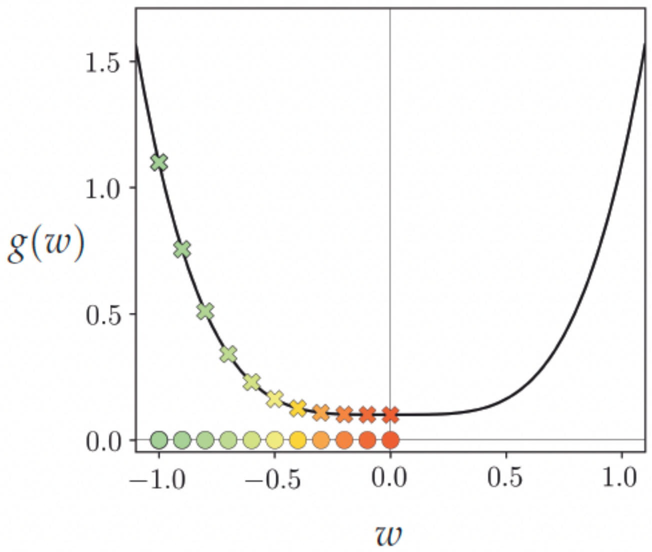
effective learing rate

$$\mathbf{w}^k = \mathbf{w}^{k-1} - \boxed{\frac{\alpha}{\left\|\nabla g(\mathbf{w}^{k-1})\right\|_2 + \epsilon}}\,\boxed{\nabla g(\mathbf{w}^{k-1})}.$$

1e-8

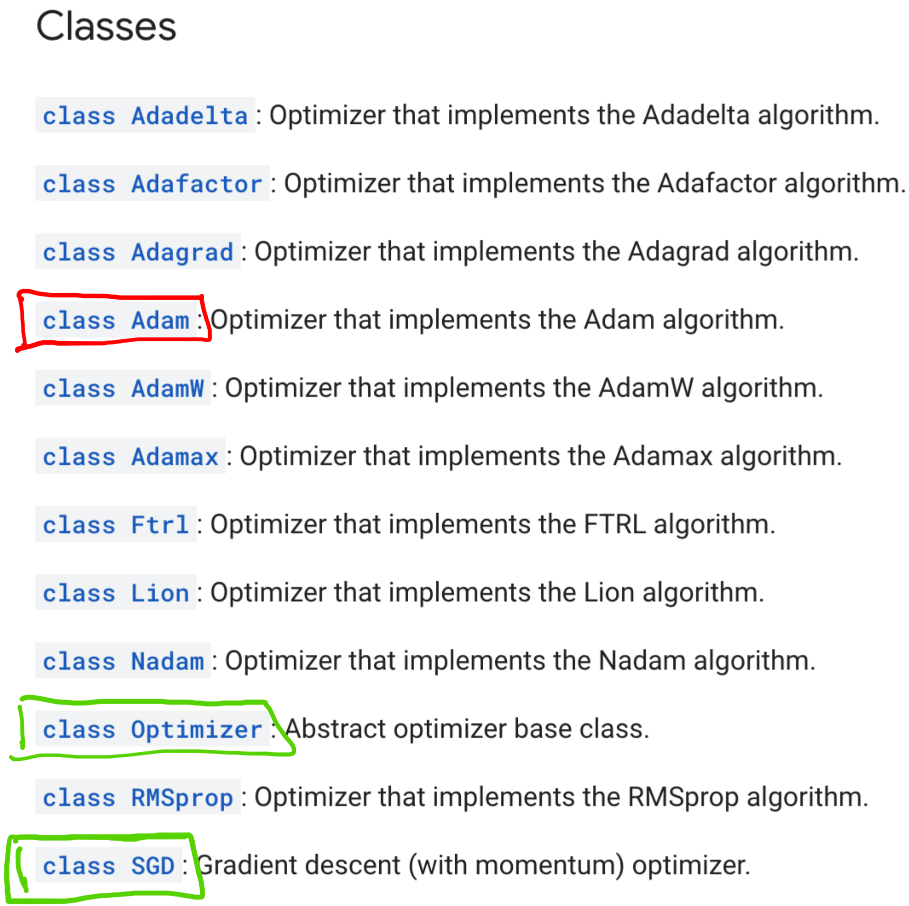
or

$$w_j^k = w_j^{k-1} - \alpha\,\frac{\frac{\partial}{\partial w_j} g\left(\mathbf{w}^{k-1}\right)}{\sqrt{\left(\frac{\partial}{\partial w_j} g\left(\mathbf{w}\right)\right)^2}}$$



Fun Stuff

https://www.tensorflow.org/api_docs/python/tf/keras/optimizers



Classes

`class Adadelta` : Optimizer that implements the Adadelta algorithm.

`class Adafactor` : Optimizer that implements the Adafactor algorithm.

`class Adagrad` : Optimizer that implements the Adagrad algorithm.

`class Adam` : Optimizer that implements the Adam algorithm.

`class AdamW` : Optimizer that implements the AdamW algorithm.

`class Adamax` : Optimizer that implements the Adamax algorithm.

`class Ftrl` : Optimizer that implements the FTRL algorithm.

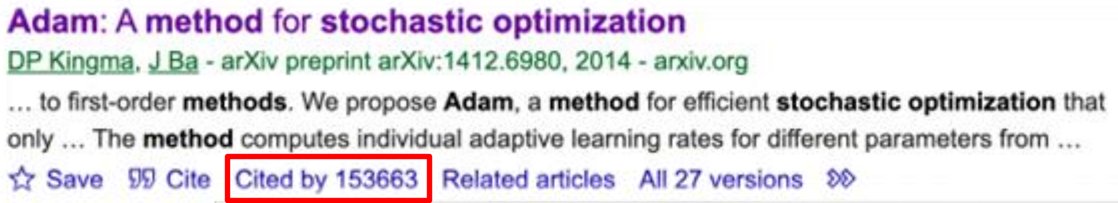
`class Lion` : Optimizer that implements the Lion algorithm.

`class Nadam` : Optimizer that implements the Nadam algorithm.

`class Optimizer` : Abstract optimizer base class.

`class RMSprop` : Optimizer that implements the RMSprop algorithm.

`class SGD` : Gradient descent (with momentum) optimizer.



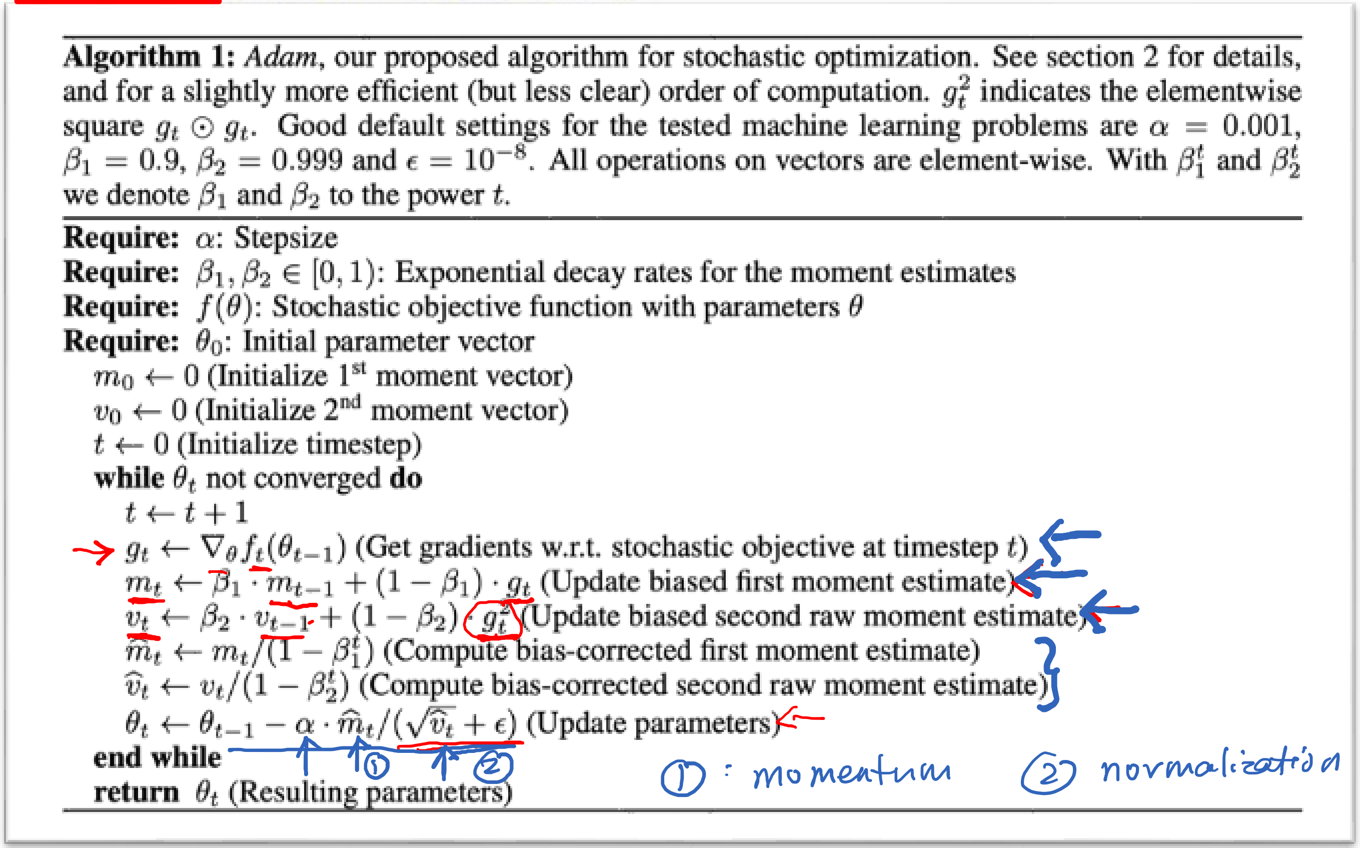
Adam: A method for stochastic optimization

DP Kingma, J Ba - arXiv preprint arXiv:1412.6980, 2014 - arxiv.org

… to first-order **methods**. We propose **Adam**, a **method** for efficient **stochastic optimization** that only … The **method** computes individual adaptive learning rates for different parameters from …

[PDF] arxiv.org

https://arxiv.org/pdf/1412.6980.pdf



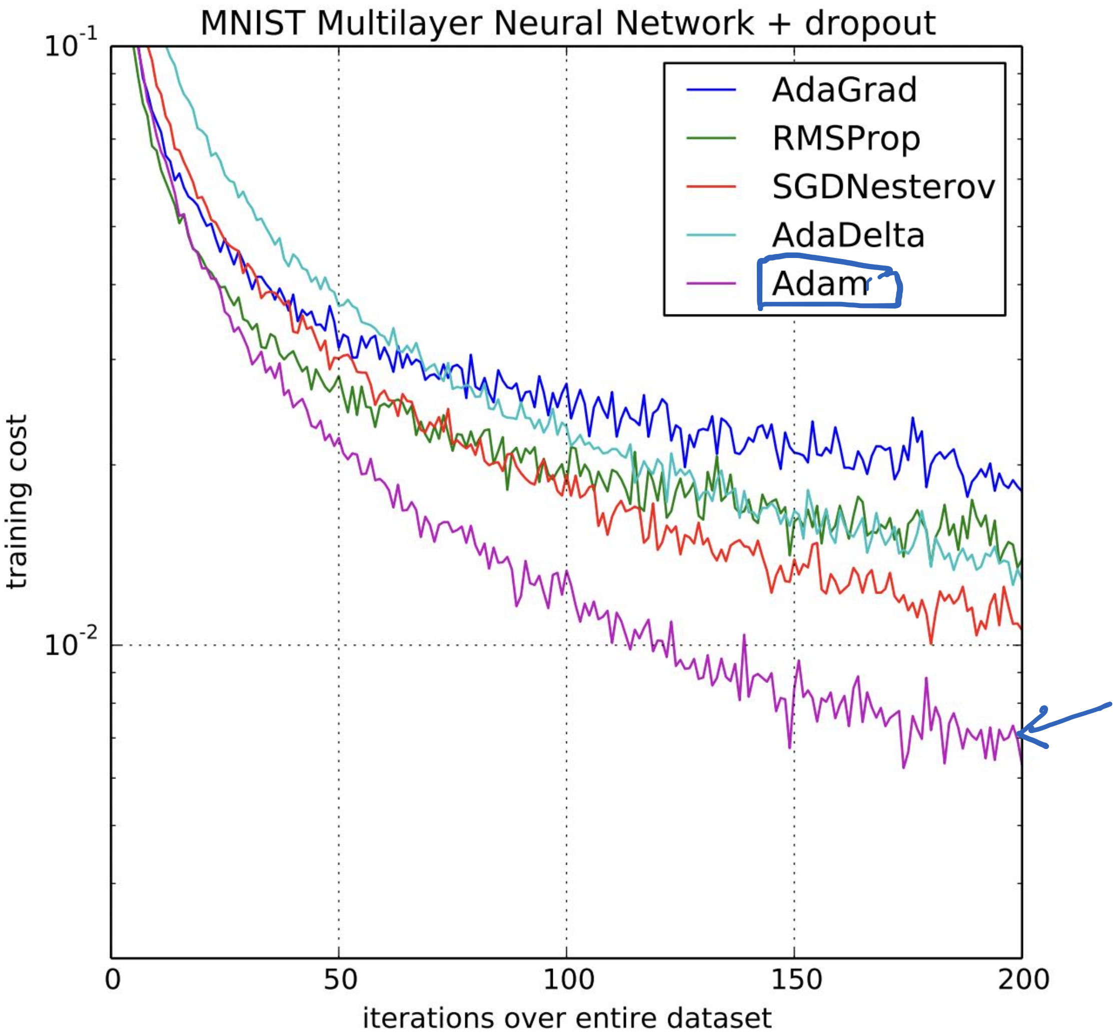
Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

```
Require: $\alpha$: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters $\theta$
Require: $\theta_0$: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return $\theta_t$ (Resulting parameters)
```

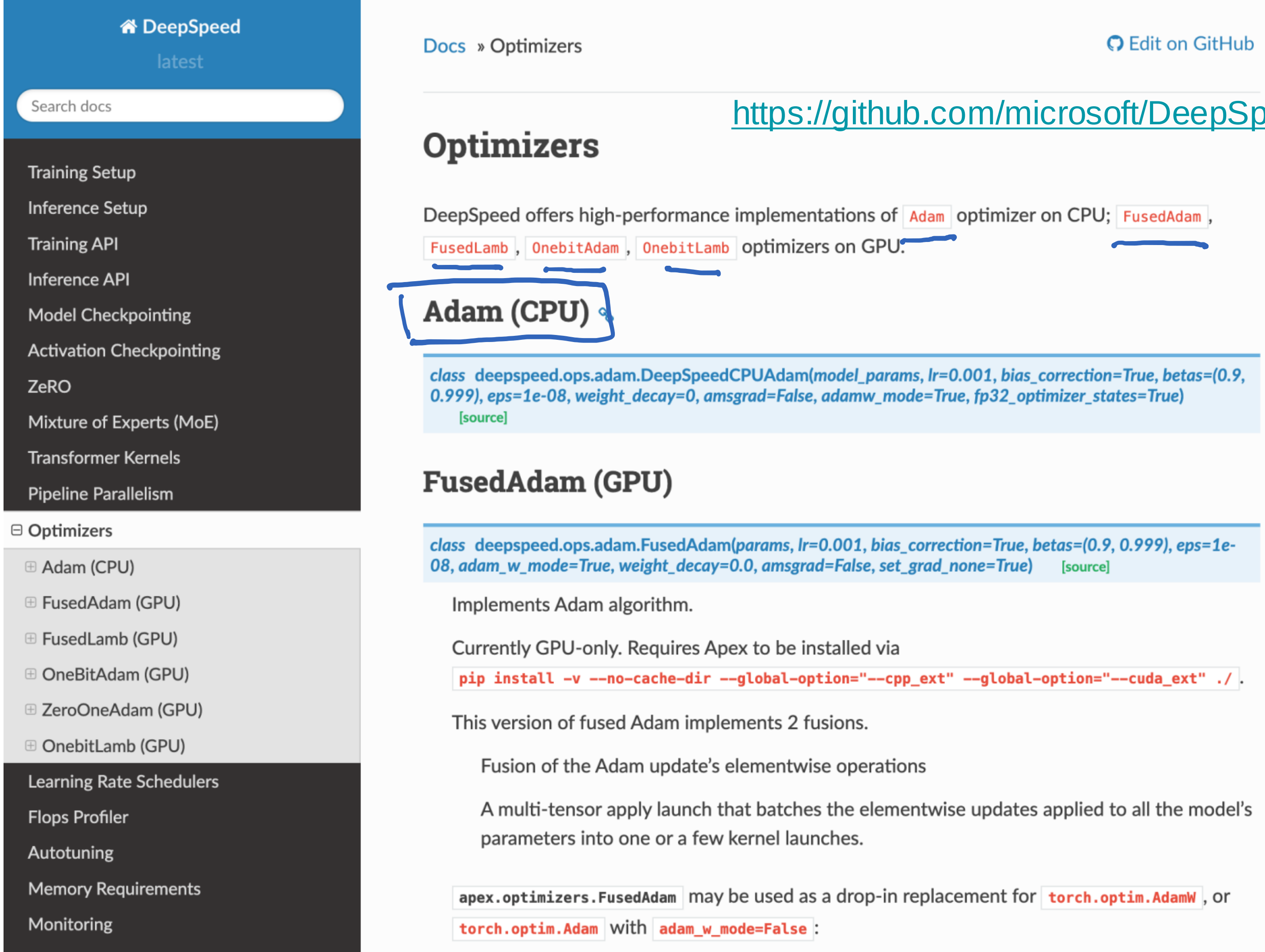
① ②

① : momentum

② normalization


MNIST Multilayer Neural Network + dropout
AdaGrad
RMSProp
SGDNesterov
AdaDelta
Adam
training cost
10^{-1}
10^{-2}
0
50
100
150
200
iterations over entire dataset

https://github.com/microsoft/DeepSpeed



Optimizers

DeepSpeed offers high-performance implementations of `Adam` optimizer on CPU; `FusedAdam`, `FusedLamb`, `OnebitAdam`, `OnebitLamb` optimizers on GPU.

Adam (CPU)

class deepspeed.ops.adam.DeepSpeedCPUAdam(*model_params, lr=0.001, bias_correction=True, betas=(0.9, 0.999), eps=1e-08, weight_decay=0, amsgrad=False, adamw_mode=True, fp32_optimizer_states=True*) [source]

FusedAdam (GPU)

class deepspeed.ops.adam.FusedAdam(*params, lr=0.001, bias_correction=True, betas=(0.9, 0.999), eps=1e-08, adam_w_mode=True, weight_decay=0.0, amsgrad=False, set_grad_none=True*) [source]

Implements Adam algorithm.

Currently GPU-only. Requires Apex to be installed via `pip install -v --no-cache-dir --global-option="--cpp_ext" --global-option="--cuda_ext" ./`.

This version of fused Adam implements 2 fusions.

> Fusion of the Adam update's elementwise operations
>
> A multi-tensor apply launch that batches the elementwise updates applied to all the model's parameters into one or a few kernel launches.

`apex.optimizers.FusedAdam` may be used as a drop-in replacement for `torch.optim.AdamW`, or `torch.optim.Adam` with `adam_w_mode=False`:

Summary

- First-order methods: use a function’s first derivatives to produce effective descent direction
- Coordinate descent
- Automatic differentiator
- Gradient decent optimization
- Weakness of gradient decent

- Concept: Stationary points, saddle points, automatic differentiator, zig-zagging, slow-crawling behavior