GRADIENT DESCENT AND GENERAL DL REMARKS

Sergey Nikolenko

Harbour Space University

June 10, 2026

Random facts:

- June 10 is the World Art Nouveau Day, celebrated on the anniversary of the death of Antoni Gaudí and Ödön Lechner
- on June 10, 1692, Bridget Bishop was hanged at Gallows Hill near Salem, Massachusetts, for "certaine Detestable Arts called Witchcraft and Sorceries"
- on June 10, 1829, the first boat race between the Universities of Oxford and Cambridge took place on the Thames; Cambridge challenged Oxford to the race, but lost easily
- on June 10, 1935, surgeon Robert Smith drank his last drink, staying sober afterwards with the help of another alcoholic, Bill Wilson; this date marks the birth of Alcoholics Anonymous
- on June 10, 1886, a government commission arrived at Neuschwanstein to formally pronounce King Ludwig II of Bavaria insane and depose him
- on June 10, 1967, Israel and Syria agreed to a cease-fire, ending the Six-Day War
- on June 10, 1987, the June Democratic Struggle started in South Korea, and people protest against the government; the demonstrations forced the military regime under president Chun Doo-hwan to institute democratic reforms leading to the Sixth Republic

Gradient descent and computational graphs

Gradient descent

- Gradient descent: take the gradient w.r.t. weights, move in that direction.
- Formally: for an error function E, targets y, and model f with parameters θ,

$$E(\theta) = \sum_{(\mathbf{x},y)\in D} E(f(\mathbf{x},\theta),y),$$

$$\theta_t = \theta_{t-1} - \eta\nabla E(\theta_{t-1}) = \theta_{t-1} - \eta \sum_{(\mathbf{x},y)\in D} \nabla E(f(\mathbf{x},\theta_{t-1}),y).$$

- So we need to sum over the entire dataset for every step?!..

Gradient descent

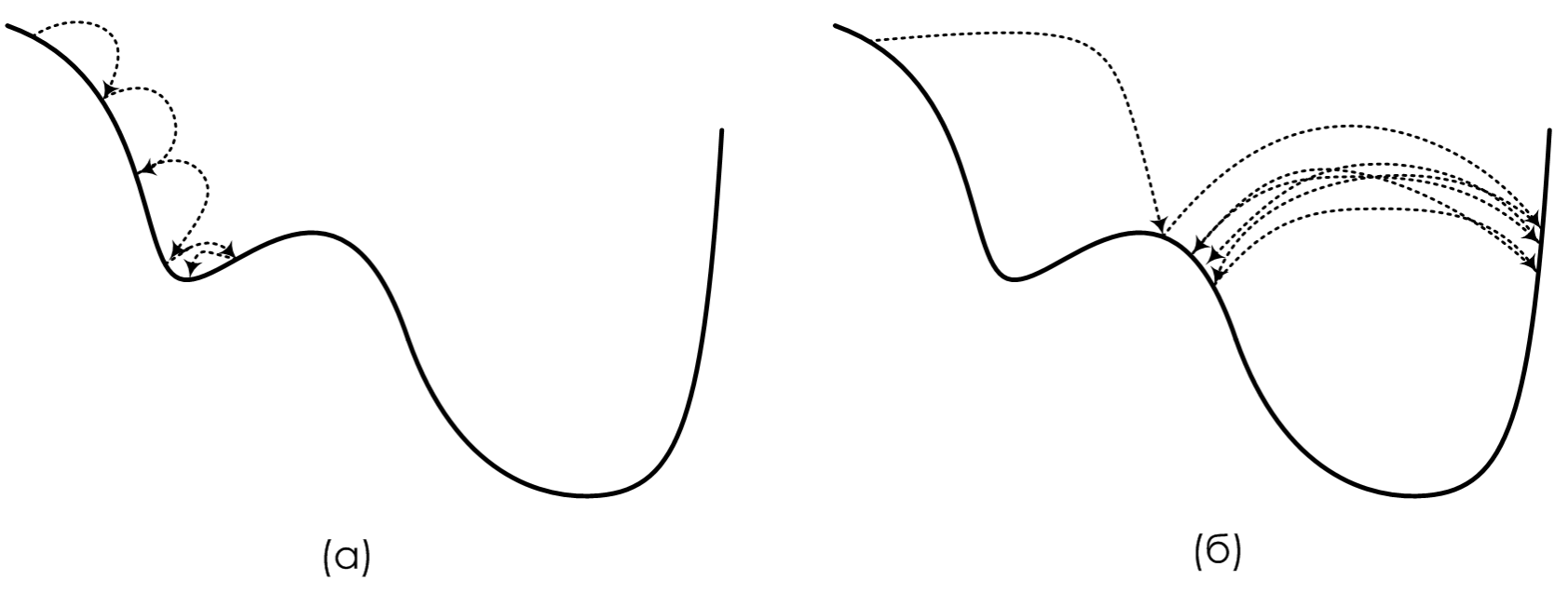
- Hence, *stochastic gradient descent*: after every training sample update

$$\theta_t = \theta_{t-1} - \eta \nabla E(f(\mathbf{x}_t, \theta_{t-1}), y_t),$$

- In practice people usually use *mini-batches*, it's easy to parallelize and smoothes out excessive "stochasticity".
- So far the only parameter is the learning rate η.

Gradient descent

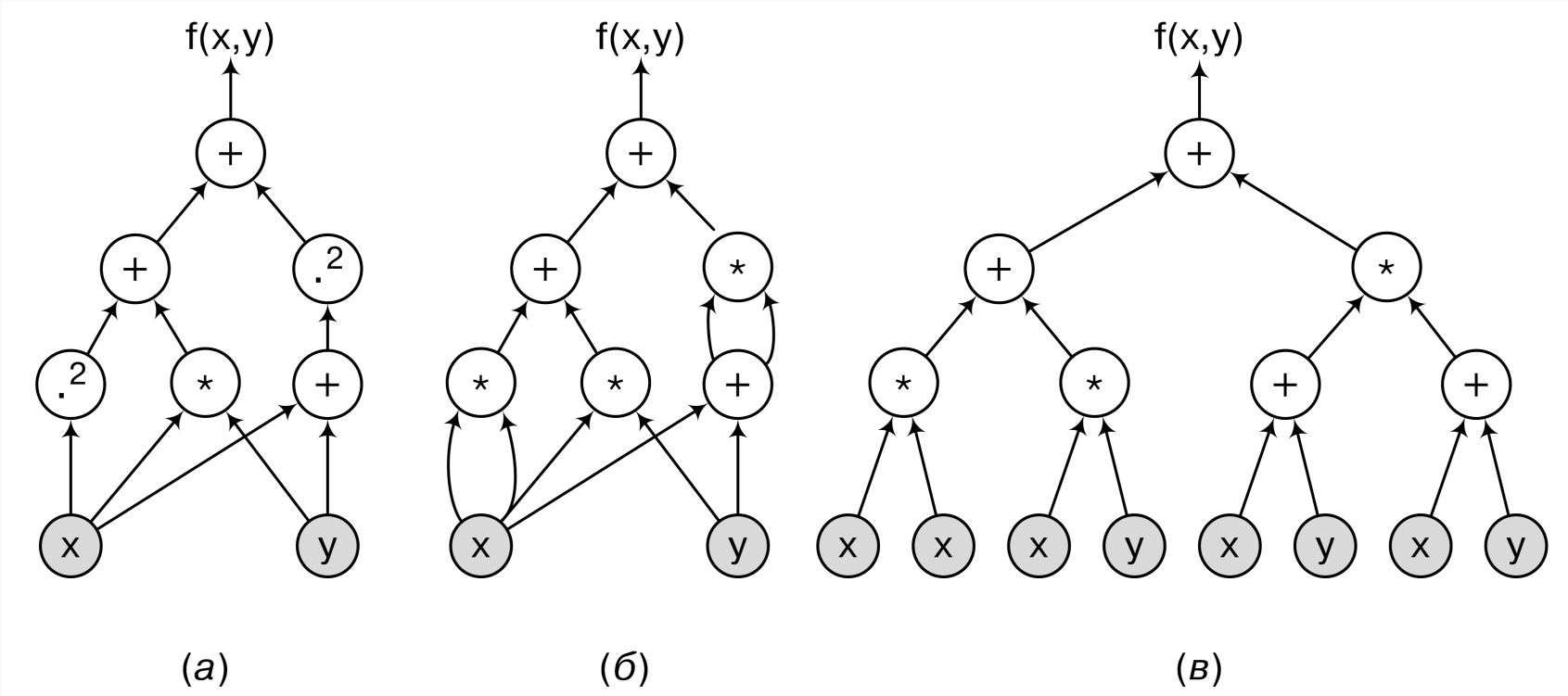
- Lots of problems with η:



- We will get to them later, for now let's concentrate on the certainly required step: the derivatives.

Computational graph, frop and bprop

- Let us represent a function as a composition of simple functions (“simple” means that we can take derivatives).
- Example – $f(x, y) = x^2 + xy + (x + y)^2$:



Computational graph, fprop and bprop

- This way we can take the gradient with the chain rule:

$$(f \circ g)'(x) = (f(g(x)))' = f'(g(x))g'(x).$$

- This simply means that an increment δx results in

$$\delta f = f'(g(x))\delta g = f'(g(x))g'(x)\delta x.$$

- We only need to be able to take gradients, i.e., derivatives w.r.t. vectors:

$$\nabla_{\mathbf{x}} f = \begin{pmatrix} \frac{\partial f}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial x_n} \end{pmatrix}.$$

$$\nabla_{\mathbf{x}}(f \circ g) = \begin{pmatrix} \frac{\partial f \circ g}{\partial x_1} \\ \vdots \\ \frac{\partial f \circ g}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \frac{\partial f}{\partial g}\frac{\partial g}{\partial x_1} \\ \vdots \\ \frac{\partial f}{\partial g}\frac{\partial g}{\partial x_n} \end{pmatrix} = \frac{\partial f}{\partial g}\nabla_{\mathbf{x}} g.$$

Computational graph, fprop and bprop

- Or, if f depends on x in several different ways, $f = f(g_1(x), g_2(x), \dots, g_k(x))$, the increment δx now comes into play several times:

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial g_1}\frac{\partial g_1}{\partial x} + \dots + \frac{\partial f}{\partial g_k}\frac{\partial g_k}{\partial x} = \sum_{i=1}^{k} \frac{\partial f}{\partial g_i}\frac{\partial g_i}{\partial x}.$$

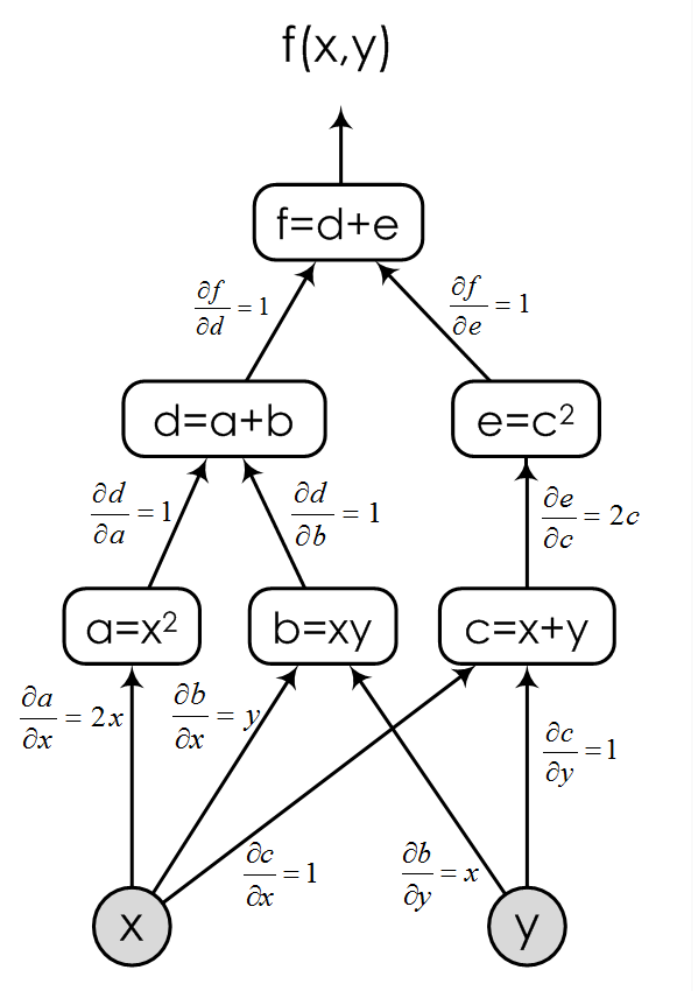
$$\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1}\nabla_{\mathbf{x}} g_1 + \dots + \frac{\partial f}{\partial g_k}\nabla_{\mathbf{x}} g_k = \sum_{i=1}^{k} \frac{\partial f}{\partial g_i}\nabla_{\mathbf{x}} g_i.$$

- Note that we got matrix multiplication for the *Jacobi matrix*:

$$\nabla_{\mathbf{x}} f = \nabla_{\mathbf{x}} \mathbf{g} \nabla_{\mathbf{g}} f, \text{ where } \nabla_{\mathbf{x}} \mathbf{g} = \begin{pmatrix} \frac{\partial g_1}{\partial x_1} & \cdots & \frac{\partial g_k}{\partial x_1} \\ \vdots & & \vdots \\ \frac{\partial g_1}{\partial x_n} & \cdots & \frac{\partial g_k}{\partial x_n} \end{pmatrix}.$$

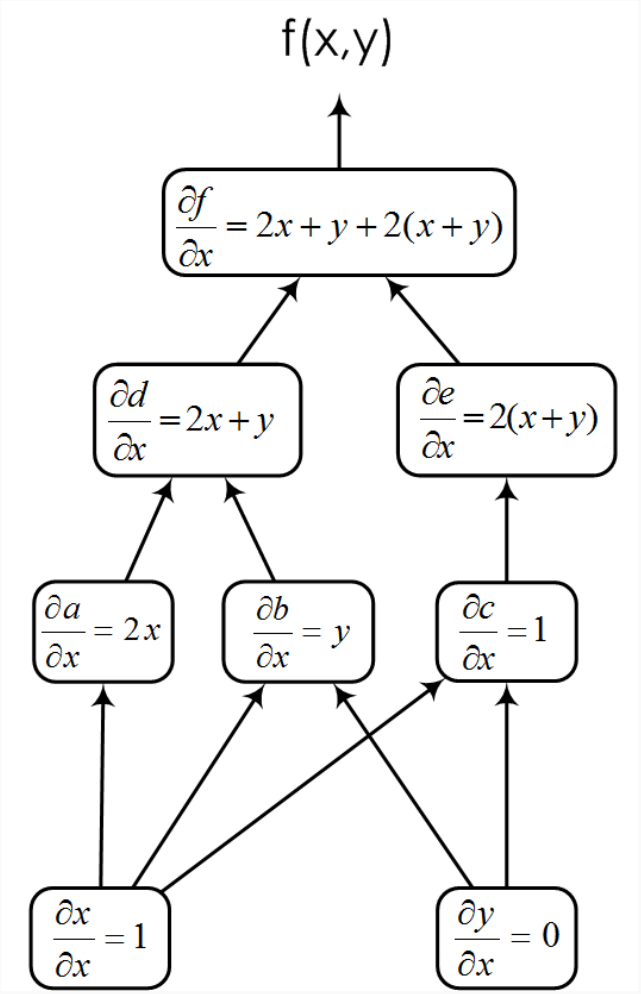
Computational graph, fprop and bprop

- Let's now go back to the example:



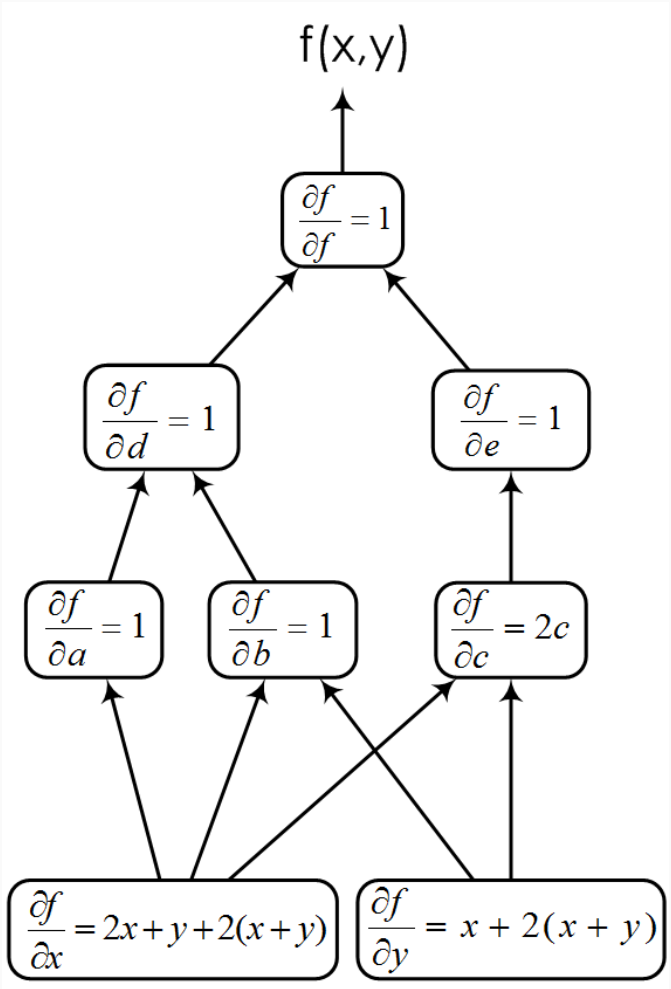
Computational graph, fprop and bprop

- Forward propagation: we compute $\frac{\partial f}{\partial x}$ by the chain rule.



Computational graph, fprop and bprop

- Backpropagation: starting from the end node, go back as $\frac{\partial f}{\partial g} = \sum_{g' \in \text{Children}(g)} \frac{\partial f}{\partial g'} \frac{\partial g'}{\partial g}$.



Computational graph, fprop and bprop

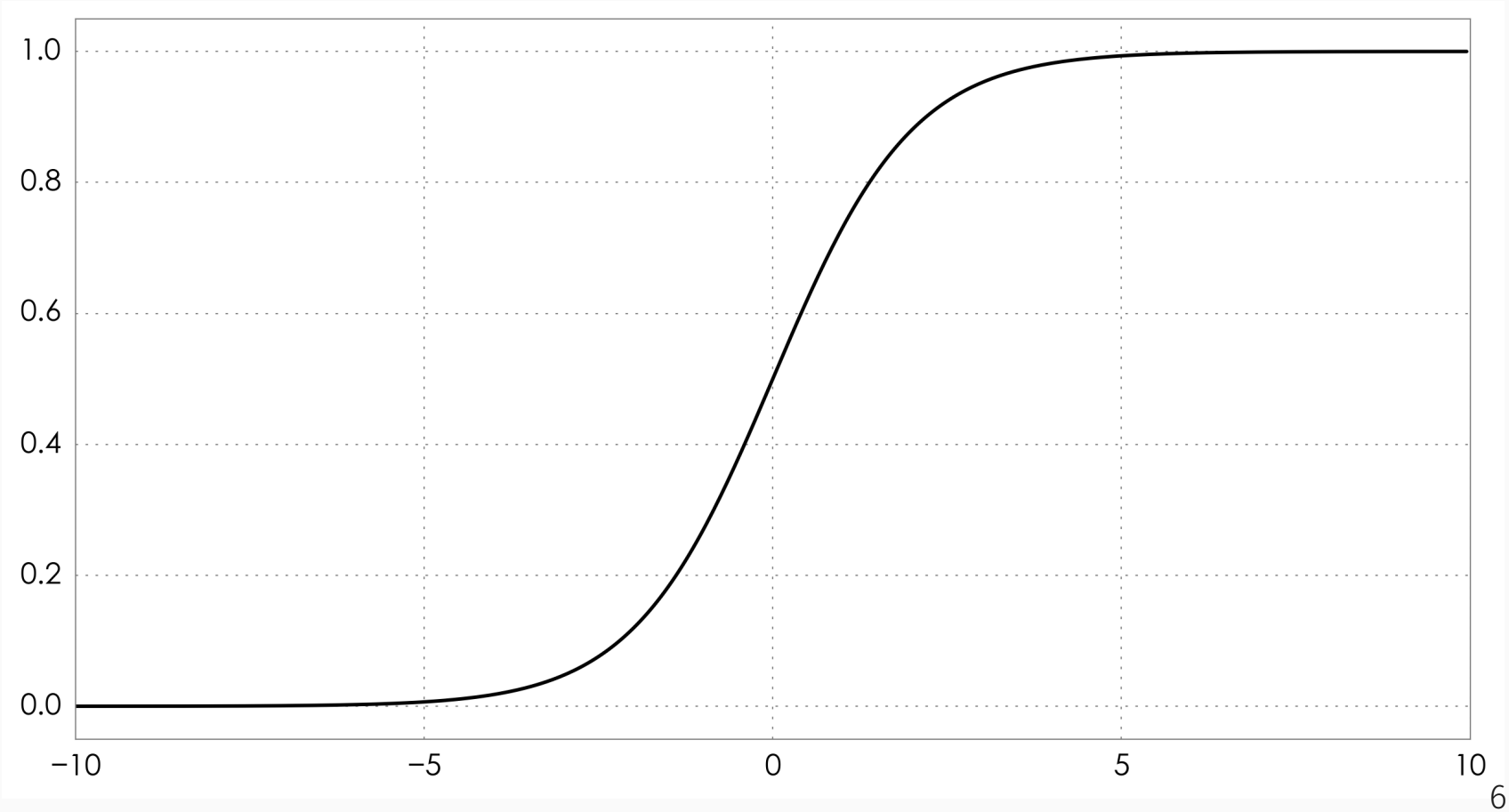
- Backprop is much better: we get all derivatives in a single pass through the graph.
- Aaaand... that's it! We can now take the gradients of any complicated composition of simple functions.
- Which is all we need to apply gradient descent!
- The libraries – *PyTorch*, *TensorFlow*, *theano* – are actually *automatic differentiation* libraries. This is their main function.
- So you can implement lots of "classical" models in *TensorFlow* and train them by gradient descent.
- And live neurons can't do that because you need two different outputs to compute the value and the derivative.

Activation functions

Activation functions

- There are many different nonlinearities; let's do a brief survey.
- Logistic sigmoid:

$$\sigma(x) = \frac{1}{1 + e^{-x}}.$$

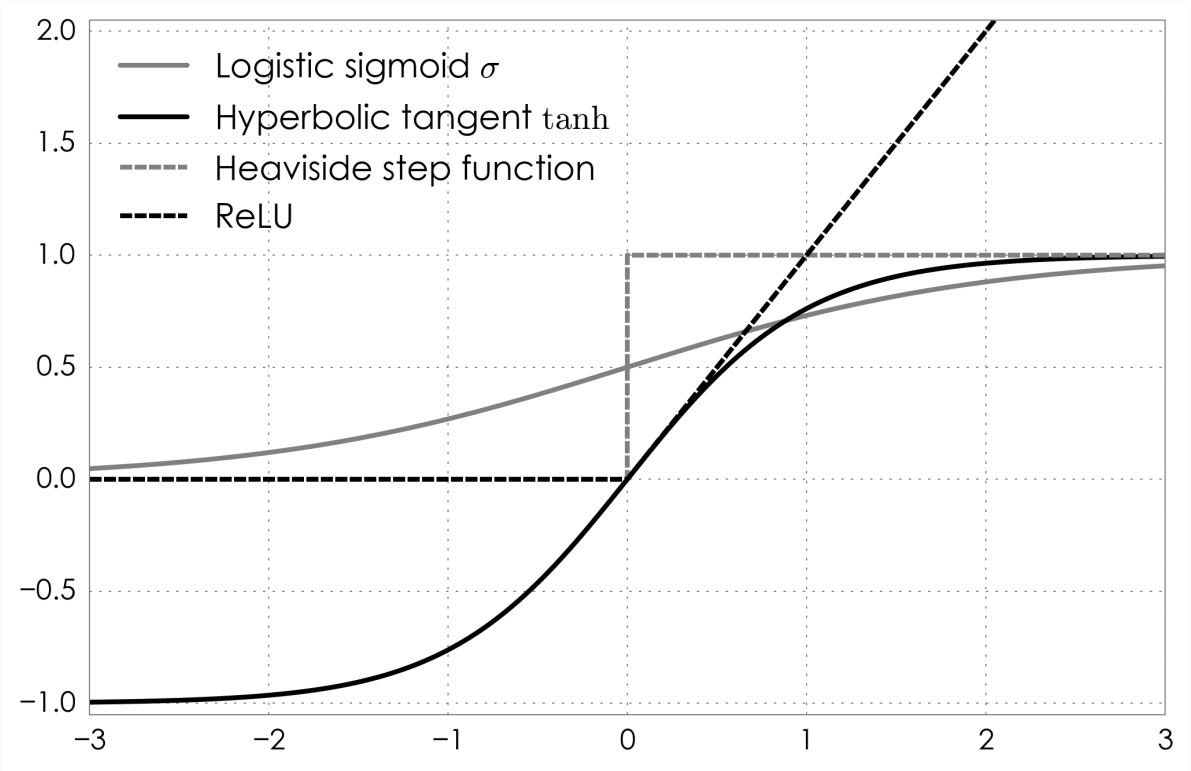


Activation functions

- Hyperbolic tangent:

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

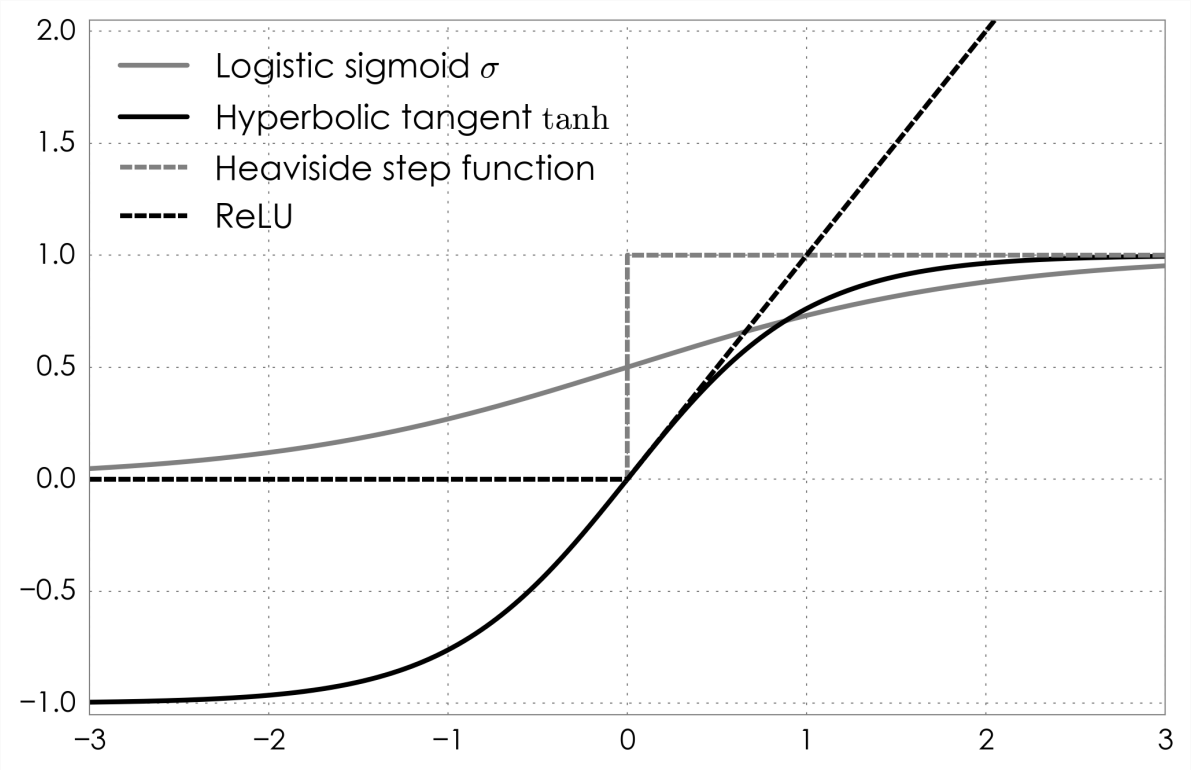
- Very similar, but zero is not a saturation point.



Activation functions

- Heaviside step function:

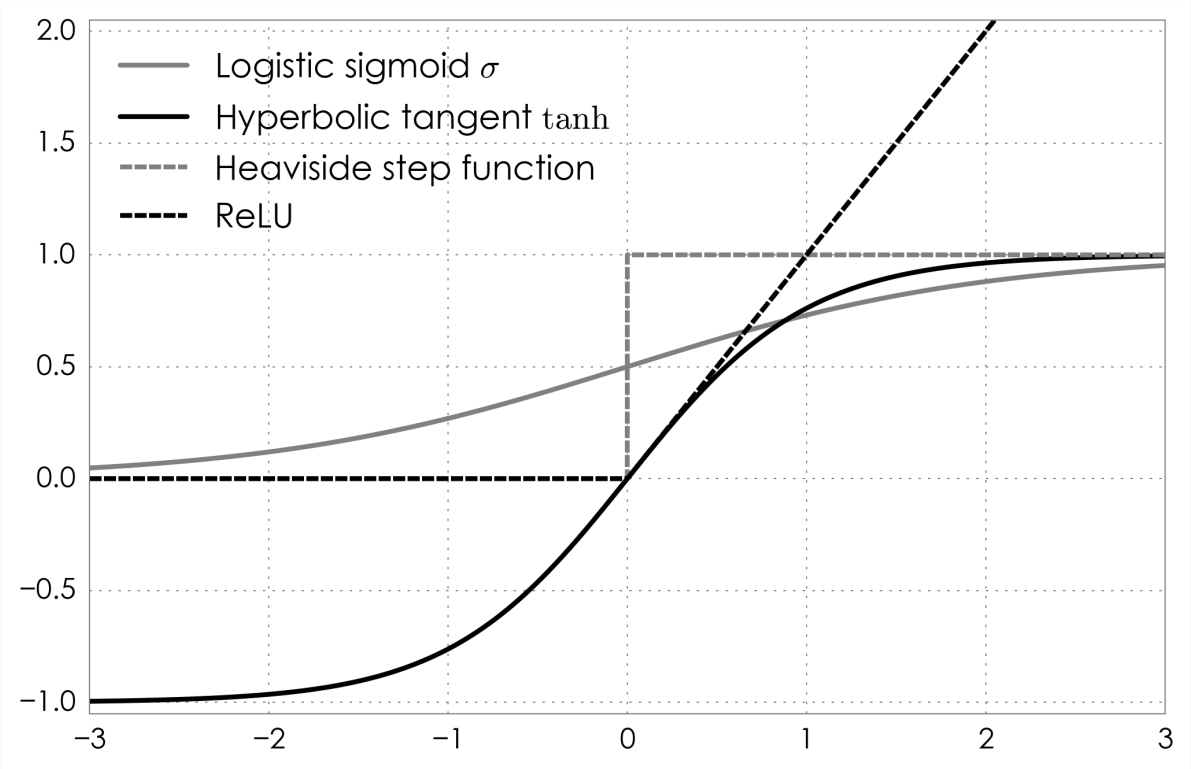
$$\text{step}(x) = \begin{cases} 0, & \text{if } x < 0, \\ 1, & \text{if } x > 0. \end{cases}$$



Activation functions

- Rectified linear unit (ReLU):

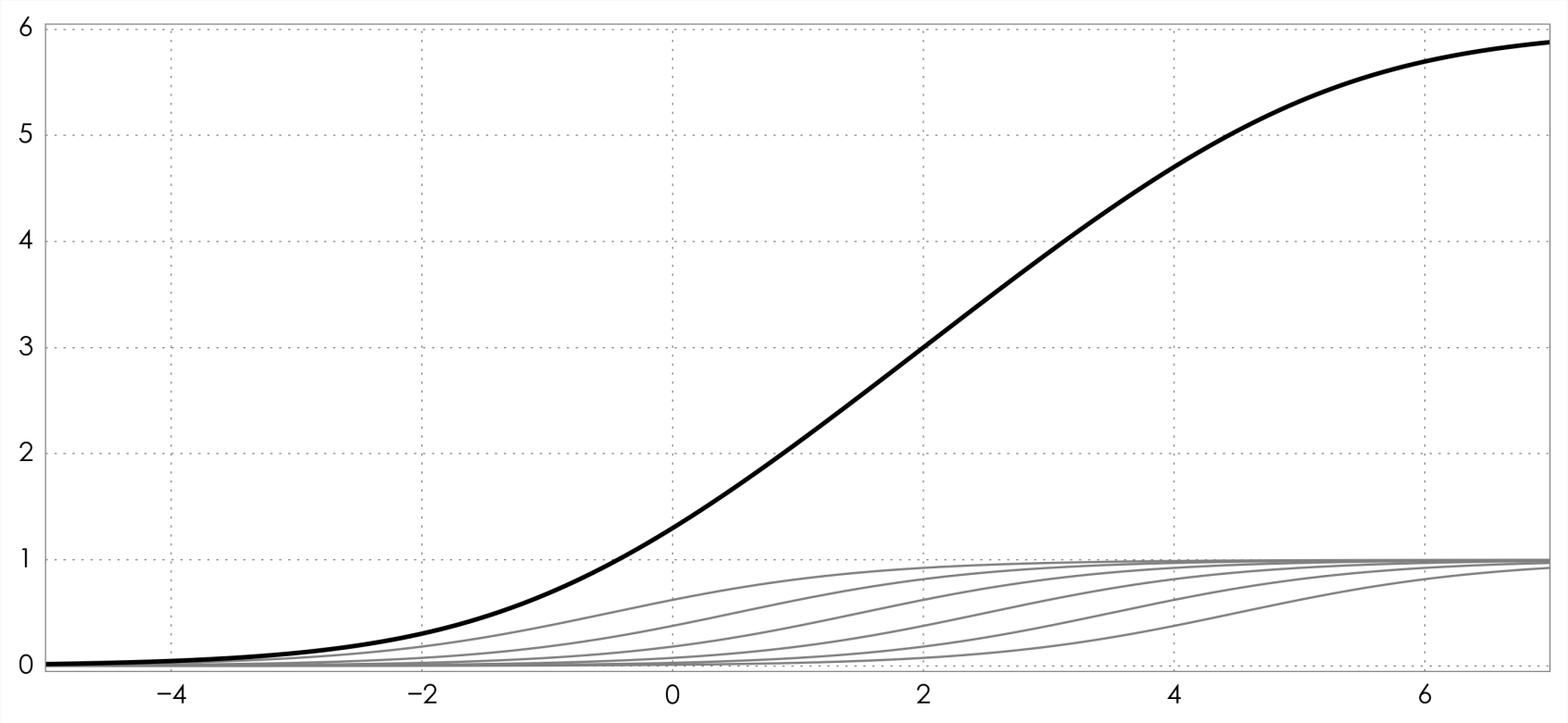
$$\mathrm{ReLU}(x) = \begin{cases} 0, & \text{if } x < 0, \\ x, & \text{if } x \geq 0. \end{cases}$$



Activation functions

- Mathematical motivation for ReLU:

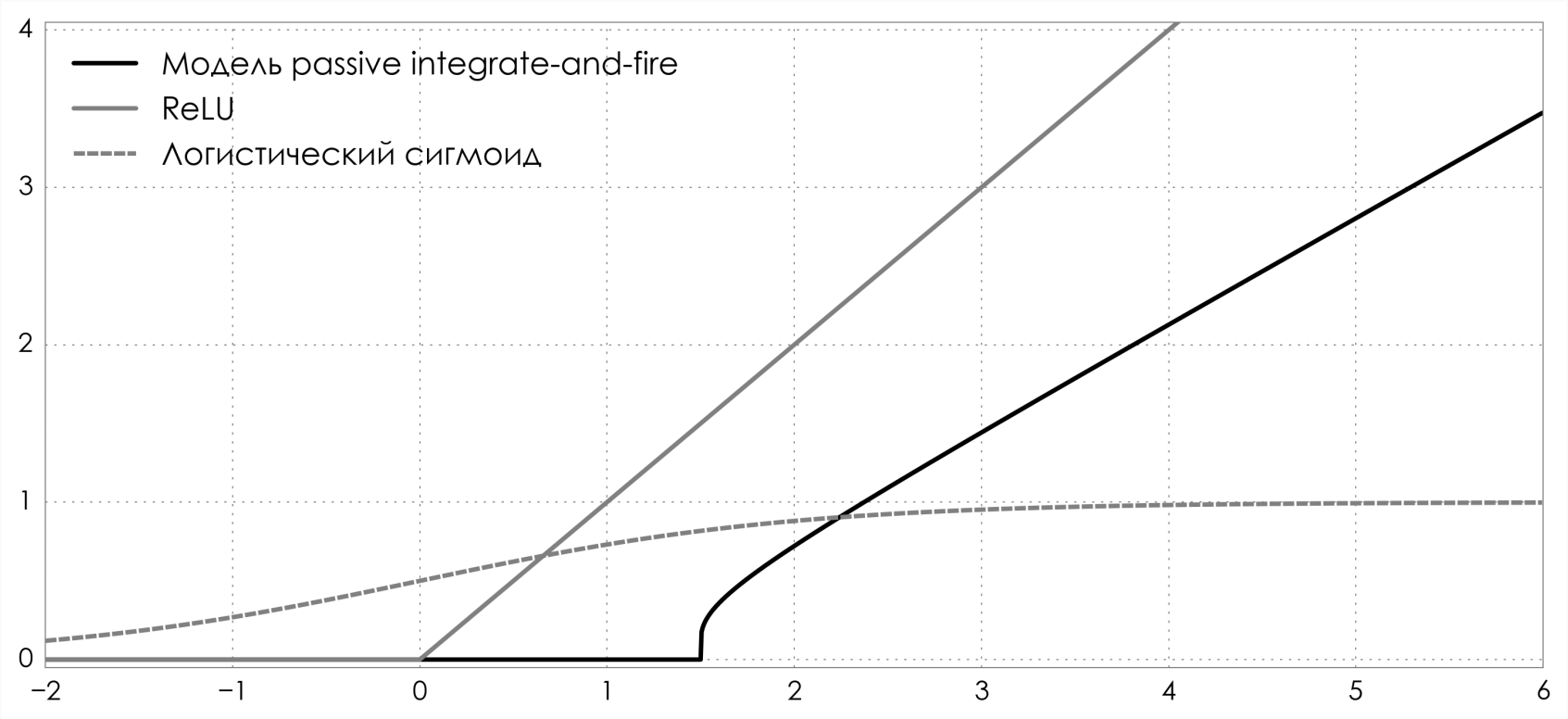
$$\sigma\left(x+\frac{1}{2}\right)+\sigma\left(x-\frac{1}{2}\right)+\sigma\left(x-\frac{3}{2}\right)+\sigma\left(x-\frac{5}{2}\right)+\ldots.$$



Activation functions

- Biological motivation for ReLU:

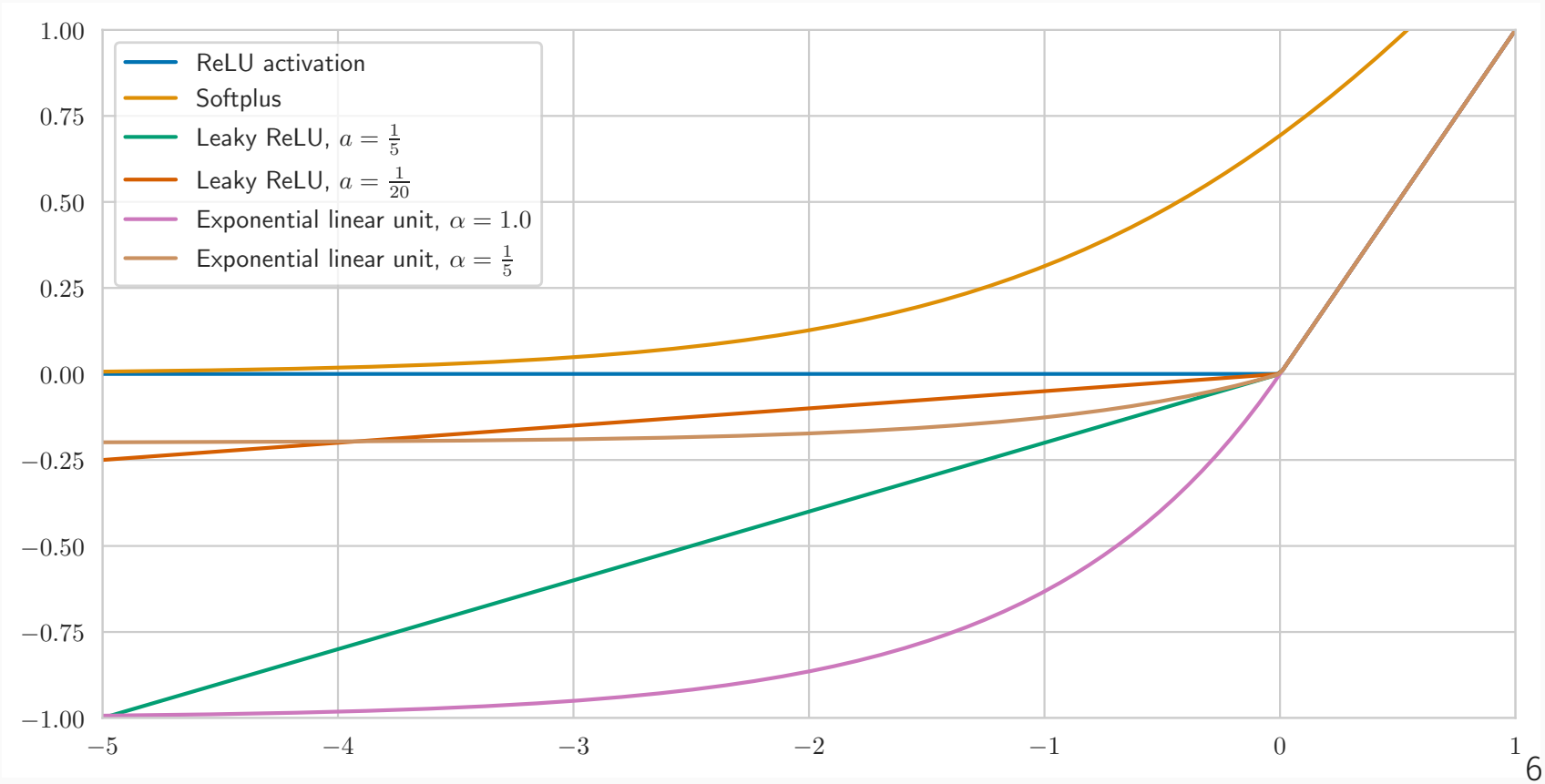
$$f(I) = \begin{cases} \left(\tau \log \frac{E+RI-V_{\text{reset}}}{E+RI-V_{th}}\right)^{-1}, & \text{if } E + RI > V_{th}, \\ 0, & \text{if } E + RI \leq V_{th}, \end{cases}$$



Activation functions

- There are several ReLU variations: $\mathrm{softplus}(x) = \ln\left(1 + e^x\right),$

$$\mathrm{LReLU}(x) = \begin{cases} ax, & \text{if } x < 0, \\ x, & \text{if } x > 0; \end{cases} \qquad \mathrm{ELU}(x) = \begin{cases} \alpha\left(e^x - 1\right), & x < 0, \\ x, & x \geq 0. \end{cases}$$

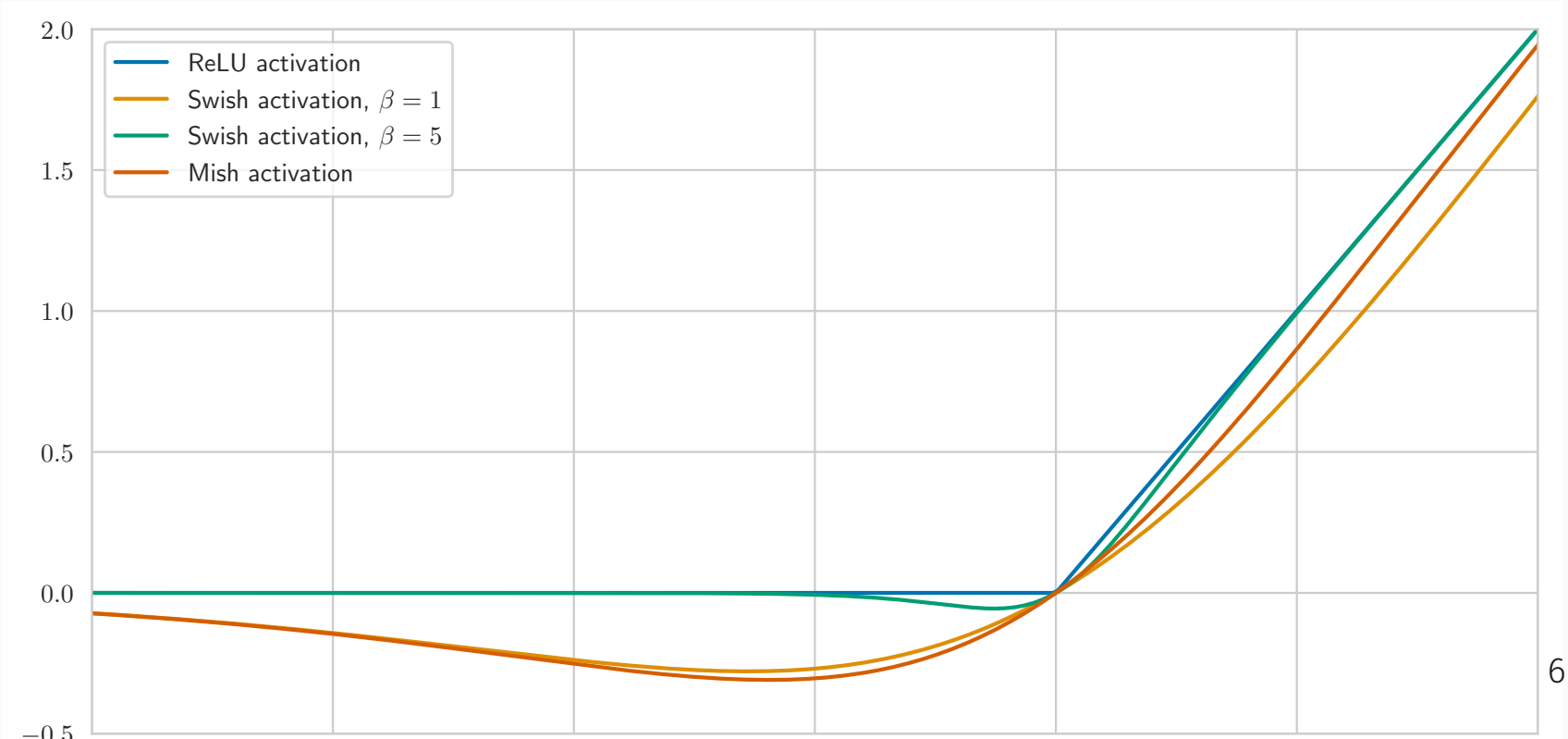


Activation functions

- But that's only the beginning. Ramachandran et al. (2017) did an automated search over the function space and found *Swish*:

$$\text{Swish}(x) = x\sigma(\beta x) = \frac{x}{1 + e^{-\beta x}}$$

- (Misra, 2019): $\text{Mish}(x) = x \tanh(\text{softplus}(x)) = x \tanh(\ln(1 + e^x))$



Activation functions

- But the most interesting development happened in 2020...
- (Ma et al., 2020): let's consider a smooth generalization of the maximum

$$S_\beta(x_1, \dots, x_n) = \frac{\sum_{i=1}^n x_i e^{\beta x_i}}{\sum_{i=1}^n e^{\beta x_i}}.$$

- And let's substitute two functions of x in there, i.e., hard maximum $\max(g(x), h(x))$ will correspond to

$$S_\beta(g(x), h(x)) = g(x)\frac{e^{\beta g(x)}}{e^{\beta g(x)} + e^{\beta h(x)}} + h(x)\frac{e^{\beta h(x)}}{e^{\beta g(x)} + e^{\beta h(x)}} =$$
$$= g(x)\sigma(\beta(g(x) - h(x))) + h(x)\sigma(\beta(h(x) - g(x))) =$$
$$= (g(x) - h(x))\,\sigma(\beta(g(x) - h(x))) + h(x).$$

Activation functions

- Ma et al. called it *ActivateOrNot* (ACON). Now this construction generalizes everything:
  - for $g(x) = x$ and $h(x) = 0$ the hard maximum is $\mathrm{ReLU}(x) = \max(x, 0)$, and the soft maximum is

$$f_{ACON-A}(x, 0) = S_\beta(x, 0) = x\sigma(\beta x),$$

    i.e., precisely *Swish*!
  - for $g(x) = x$ and $h(x) = ax$ for some $a < 1$, the hard maximum is $\mathrm{LReLU}(x) = \max(x, px)$, and the soft maximum is

$$f_{ACON-B} = S_\beta(x, ax) = (1 - a)\, x\sigma(\beta(1 - a)x) + ax;$$

- Ma et al. called it *ActivateOrNot* (ACON). Now this construction generalizes everything:
  - and this can be straightforwardly generalized to

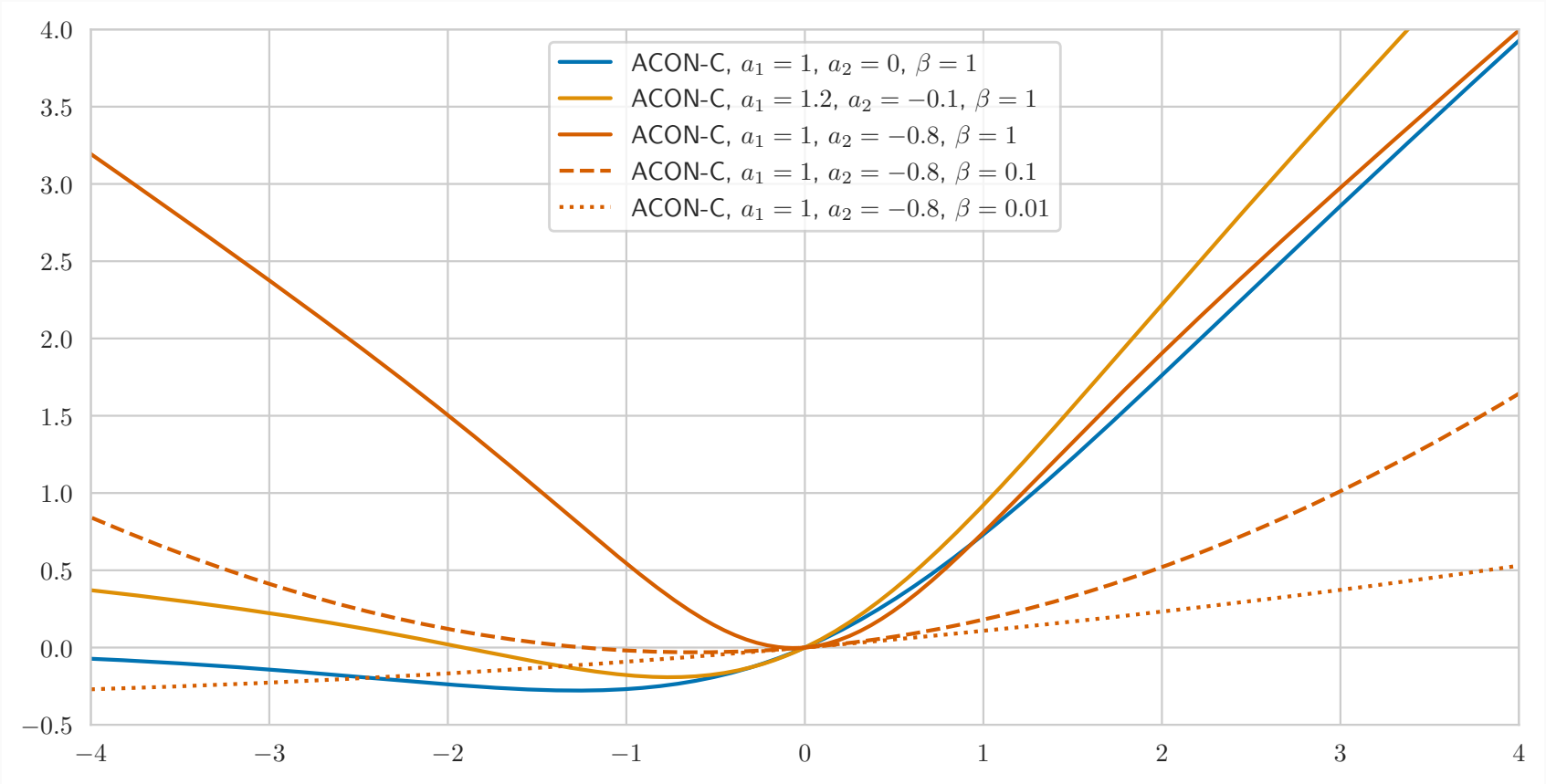
$$f_{ACON-C} = S_\beta(a_1 x, a_2 x) = (a_1 - a_2)\, x\sigma\left(\beta(a_1 - a_2)x\right) + a_2 x;$$

now a_1 and a_2 can be learnable parameters, and intuitively it's just the limits of the derivative on both sides:

$$\lim_{x\to\infty} \frac{\mathrm{d}f_{ACON-C}}{\mathrm{d}x} = a_1, \qquad \lim_{x\to-\infty} \frac{\mathrm{d}f_{ACON-C}}{\mathrm{d}x} = a_2.$$

Activation functions

- Here is ACON-C for different parameter values:



Activation functions

- Ma et al. show that by tuning the parameters one can win a lot even in bog-standard tasks:

	ReLU			meta-ACON		
	FLOPs	# Params.	Top-1 err.	FLOPs	# Params.	Top-1 err.
MobileNetV1 0.25	41M	0.5M	47.6	41M	0.6M	**40.9**$_{(+6.7)}$
MobileNetV2 0.17	42M	1.4M	52.6	42M	1.9M	**46.2**$_{(+6.4)}$
ShuffleNetV2 0.5x	41M	1.4M	39.4	41M	1.7M	**34.8**$_{(+4.6)}$
MobileNetV1 0.75	325M	2.6M	30.2	326M	3.1M	**26.4**$_{(+3.8)}$
MobileNetV2 1.0	299M	3.5M	27.9	299M	6.0M	**25.0**$_{(+2.9)}$
ShuffleNetV2 1.5x	299M	3.5M	27.4	299M	3.9M	**24.7**$_{(+2.7)}$
ResNet-18	1.8G	11.7M	30.3	1.8G	11.9M	**28.4**$_{(+1.9)}$
ResNet-50	3.9G	25.5M	24.0	3.9G	25.7M	**22.0**$_{(+2.0)}$
ResNet-101	7.6G	44.4M	22.8	7.6G	44.8M	**21.0**$_{(+1.8)}$
ResNet-152	11.3G	60.0M	22.3	11.3G	60.5M	**20.5**$_{(+1.8)}$

- I haven't seen much of ACON-C in actual practice, but this just goes to show that deep learning has a lot of not-very-hidden depths even in the simplest things.

Variations of gradient descent

Gradient descent

- "Vanilla" gradient descent:

$$\mathbf{x}_k = \mathbf{x}_{k-1} - \alpha \nabla f(\mathbf{x}_k).$$

- So much depends on the learning rate α.
- First idea — let α decrease with time:
  - linear decay:

$$\alpha = \alpha_0 \left(1 - \frac{t}{T}\right);$$

  - exponential decay:

$$\alpha = \alpha_0 e^{-\frac{t}{T}}.$$

Gradient descent

- There're a lot of results here. Wolfe conditions: if we are solving $\min_{\mathbf{x}} f(\mathbf{x})$, and on step k we already know the direction $\mathbf{p}_k$ where to go (e.g., $\mathbf{p}_k = \nabla_{\mathbf{x}} f(\mathbf{x}_k)$), i.e., we need to solve $\min_{\alpha} f(\mathbf{x}_k + \alpha \mathbf{p}_k)$, then:
  - for $\phi_k(\alpha) = f(\mathbf{x}_k + \alpha \mathbf{p}_k)$ we have $\phi_k'(\alpha) = \nabla f(\mathbf{x}_k + \alpha \mathbf{p}_k)^\top \mathbf{p}_k$, and if $\mathbf{p}_k$ is the descent direction then $\phi_k'(0) < 0$;
  - the step size α must satisfy the Armijo rule:

$$\phi_k(\alpha) \leq \phi_k(0) + c_1 \alpha \phi_k'(0) \text{ for some } c_1 \in (0, \frac{1}{2});$$

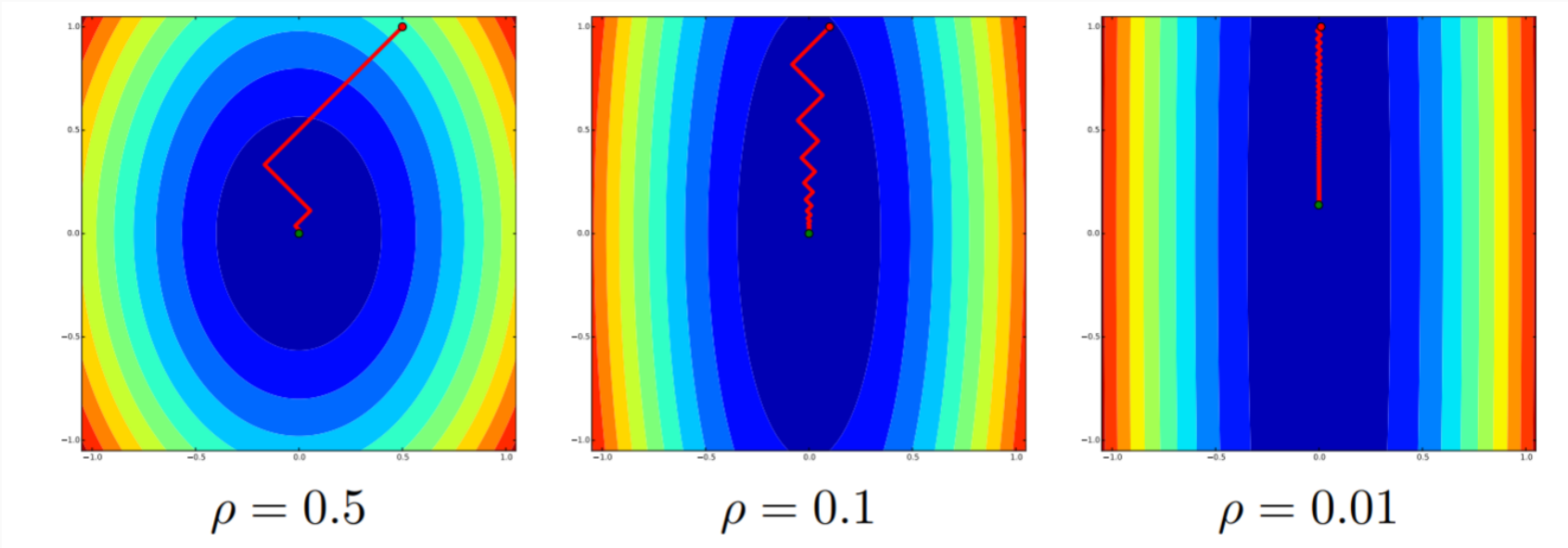
  - or even stronger Wolfe's rule: Armijo rule plus

$$|\phi_k'(\alpha)| \leq c_2 |\phi_k'(0)|,$$

    i.e., we'd like to decrease the projection of the gradient.
- We stop the process when $\|\nabla_{\mathbf{x}} f(\mathbf{x}_k)\|^2 \leq \epsilon$ or $\|\nabla_{\mathbf{x}} f(\mathbf{x}_k)\|^2 \leq \epsilon \|\nabla_{\mathbf{x}} f(\mathbf{x}_0)\|^2$ (why square, btw?).

Gradient descent

- Let's see what happens if the scale is different: for a function $f(x, y) = \frac{1}{2}x^2 + \frac{\rho}{2}y^2 \to \min_{x,y}$



- For elongated "valleys" (variables with different scale) we get a lot of superfluous iterations, everything is very slow.
- Much better to be *adaptive*; but how?

Gradient descent

- The best thing in life is, of course, *Newton's method*: let's scale back with the Hessian

$$\mathbf{g}_k = \nabla_{\mathbf{x}} f(\mathbf{x}_k),\ H_k = \nabla^2_{\mathbf{x}} f(\mathbf{x}_k),\ \text{and } \mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k H_k^{-1} \mathbf{g}_k.$$

- Armijo rule is applicable here as well:

$$\alpha_k: \quad f(\mathbf{x}_{k+1}) \leq f(\mathbf{x}_k) - c_1 \alpha_k g_k^\top H_k^{-1} \mathbf{g}_k,\ c_1 \approx 10^{-4}.$$

- Would be very nice but, alas, you can't just compute H_k.

Gradient descent

- There are approximations.
- Conjugate gradients, quasi-Newtonian methods...
- L-BFGS (limited memory Broyden–Fletcher–Goldfarb–Shanno):
  - construct an approximation to H^{-1};
  - to do that, save updates of the arguments and then express H^{-1} through them.
- Important open question: can we make L-BFGS or something similar work for deep learning?
- Doesn't work so far, mostly because you do need to be able to compute the gradient.
- And we aren't. Wait, what?..

Stochastic gradient descent

- We are usually dealing with stochastic gradient descent:

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \alpha \nabla f(\mathbf{x}_t, \mathbf{x}_{t-1}, y_t).$$

- With mini-batches, too. How do we understand this formally?
- We are usually dealing with *stochastic optimization* problems:

$$F(\mathbf{x}) = \mathbb{E}_{q(\mathbf{y})} f(\mathbf{x}, \mathbf{y}) \to \min_{\mathbf{x}}:$$

  - empirical risk minimization:

$$F(\mathbf{x}) = \frac{1}{N} \sum_{i=1}^{N} f_i(\mathbf{x}) = \mathbb{E}_{i \sim \mathrm{U}(1,\dots,N)} f_i(\mathbf{x}) \to \min_{\mathbf{x}};$$

  - variational lower bound (ELBO) minimization... but later about that (if ever).

- So what are mini-batches, then?

Stochastic gradient descent

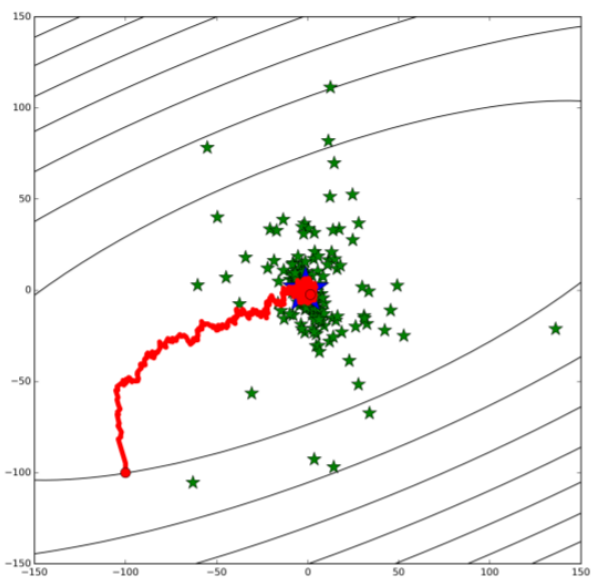
- Simply empirical estimates of a random function from a sample:

$$\hat{F}(\mathbf{x}) = \frac{1}{m}\sum_{i=1}^{m} f(\mathbf{x}, \mathbf{y}_i), \quad \hat{\mathbf{g}}(\mathbf{x}) = \frac{1}{m}\sum_{i=1}^{m} \nabla_{\mathbf{x}} f(\mathbf{x}, \mathbf{y}_i).$$

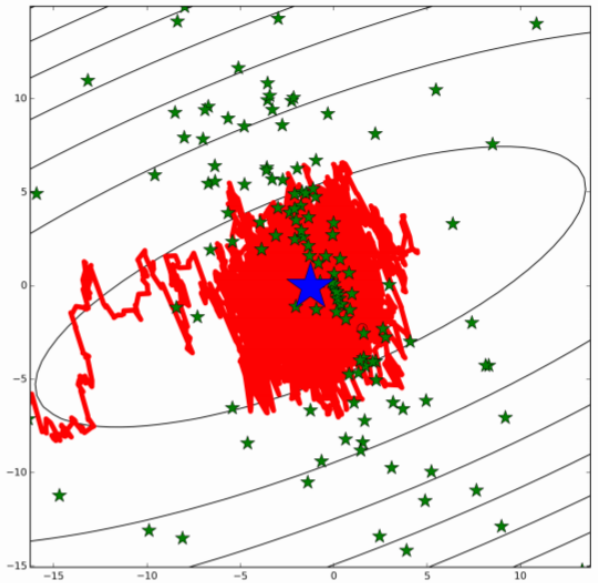
- Very good estimates: unbiased, converging (albeit slowly), easy to compute.
- In general, that's how you motivate stochastic gradient descent (SGD), it's a Monte-Carlo approach. But there are problems...

Stochastic gradient descent

- SGD problems:
  - never goes in the right direction,
  - even at the exact optimum of $F(\mathbf{x})$ the step is nonzero, i.e., it cannot converge with a constant step size,
  - we know neither $F(\mathbf{x})$ nor $\nabla F(\mathbf{x})$, i.e., we cannot use Armijo and Wolfe's rules.



SGD trajectory



optimum vicinity

Stochastic gradient descent

- Nevertheless, we can try to analyze an SGD iteration for $F(\mathbf{x}) = \mathbb{E}_{q(\mathbf{y})} f(\mathbf{x}, \mathbf{y}) \to \min_{\mathbf{x}}$:

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \alpha_k \hat{\mathbf{g}}_k, \quad \mathbb{E}\hat{\mathbf{g}}_k = \mathbf{g}_k = \nabla F(\mathbf{x}_k).$$

- Let's estimate the residue of a point on some iteration:

$$\begin{aligned}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 &= \|\mathbf{x}_k - \alpha_k \hat{\mathbf{g}}_k - \mathbf{x}_{\text{opt}}\|^2 = \\ &= \|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 - 2\alpha_k \hat{\mathbf{g}}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) + \alpha_k^2 \|\hat{\mathbf{g}}_k\|^2.\end{aligned}$$

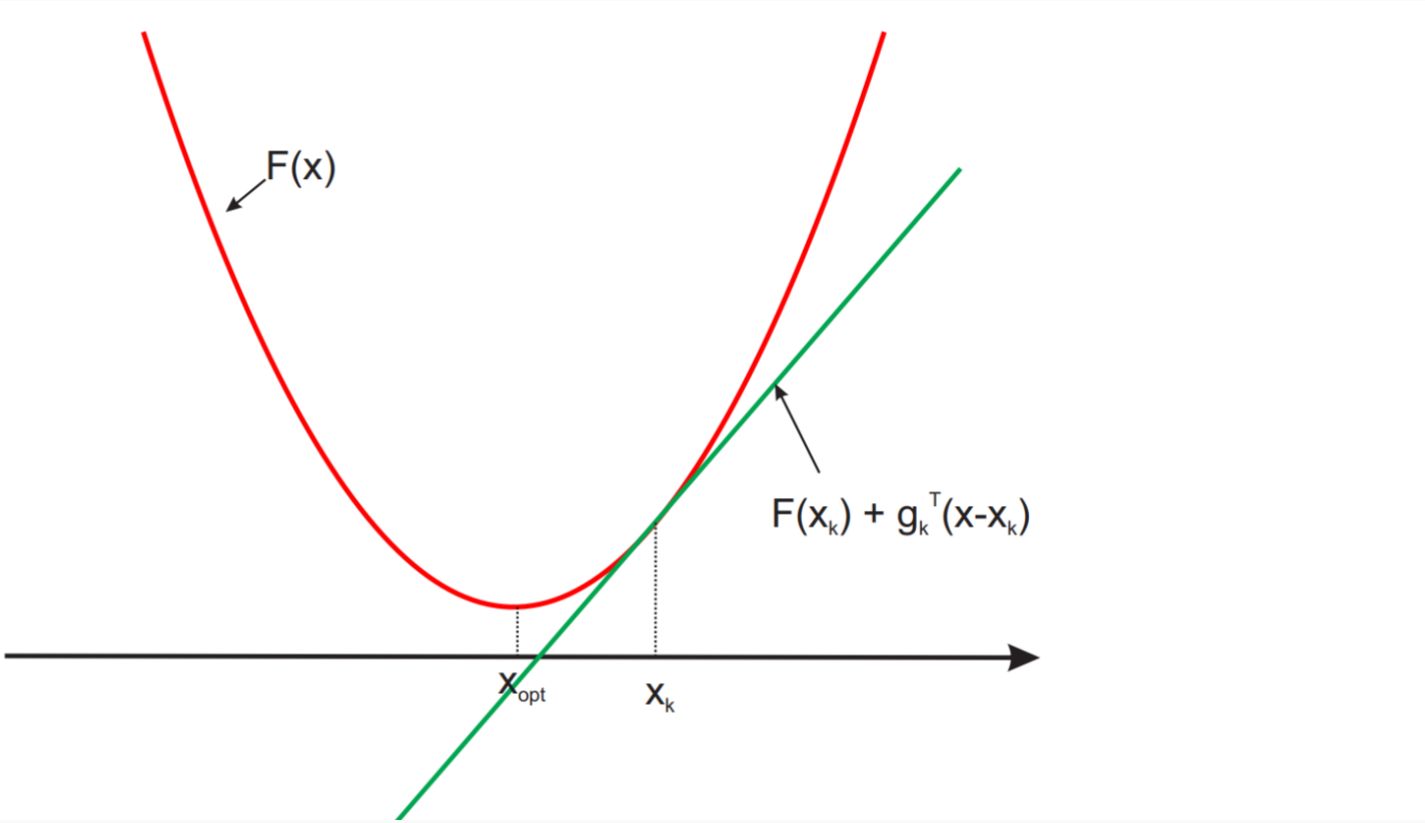
- Take expectation with respect to $q(\mathbf{y})$ at time moment k:

$$\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 = \|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 - 2\alpha_k \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) + \alpha_k^2 \mathbb{E}\|\hat{\mathbf{g}}_k\|^2.$$

Stochastic gradient descent

- Assume for simplicity that F is convex:

$$F(\mathbf{x}_{\text{opt}}) \geq F(\mathbf{x}_k) + \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}})$$



Stochastic gradient descent

- We had

$$\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 = \|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 - 2\alpha_k \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) + \alpha_k^2 \mathbb{E}\|\hat{\mathbf{g}}_k\|^2,$$
$$F(\mathbf{x}_{\text{opt}}) \geq F(\mathbf{x}_k) + \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}).$$

- Therefore,

$$\alpha_k (F(\mathbf{x}_k) - F(\mathbf{x}_{\text{opt}})) \leq \alpha_k \mathbf{g}_k^\top (\mathbf{x}_k - \mathbf{x}_{\text{opt}}) =$$
$$= \frac{1}{2}\|\mathbf{x}_k - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2}\alpha_k^2 \mathbb{E}\|\hat{\mathbf{g}}_k\|^2 - \frac{1}{2}\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2.$$

Stochastic gradient descent

- Take the expectation of the left-hand side and sum up:

$$\sum_{i=0}^{k} \alpha_i (\mathbb{E}F(\mathbf{x}_i) - F(\mathbf{x}_{\text{opt}})) \leq$$

$$\leq \frac{1}{2}\|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2}\sum_{i=0}^{k} \alpha_i^2 \mathbb{E}\|\hat{\mathbf{g}}_i\|^2 - \frac{1}{2}\mathbb{E}\|\mathbf{x}_{k+1} - \mathbf{x}_{\text{opt}}\|^2 \leq$$

$$\leq \frac{1}{2}\|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2}\sum_{i=0}^{k} \alpha_i^2 \mathbb{E}\|\hat{\mathbf{g}}_i\|^2.$$

- We got the sum of function values at different points with weights α_i. What do we do now?

Stochastic gradient descent

- Use convexity:

$$\mathbb{E}F\left(\frac{\sum_i \alpha_i \mathbf{x}_i}{\sum_i \alpha_i}\right) - F(\mathbf{x}_{\text{opt}}) \leq$$

$$\leq \frac{\sum_i \alpha_i (\mathbb{E}F(\mathbf{x}_i) - F(\mathbf{x}_{\text{opt}})}{\sum_i \alpha_i} \leq \frac{\frac{1}{2}\|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\|^2 + \frac{1}{2}\sum_{i=0}^k \alpha_i^2 \mathbb{E}\|\hat{\mathbf{g}}_i\|^2}{\sum_i \alpha_i}.$$

- The estimate is on the value in a linear combination of points (in practice there is no difference or it's even better to take the last point).
- If $\|\mathbf{x}_0 - \mathbf{x}_{\text{opt}}\| \leq R$ and $\mathbb{E}\|\hat{\mathbf{g}}_k\|^2 \leq G^2$ then

$$\mathbb{E}F(\hat{\mathbf{x}}_k) - F(\mathbf{x}_{\text{opt}}) \leq \frac{R^2 + G^2 \sum_{i=0}^k \alpha_i^2}{2\sum_{i=0}^k \alpha_i}.$$

Stochastic gradient descent

- The main bound regarding SGD:

$$\mathbb{E}F(\hat{\mathbf{x}}_k) - F(\mathbf{x}_{\text{opt}}) \leq \frac{R^2 + G^2 \sum_{i=0}^{k} \alpha_i^2}{2 \sum_{i=0}^{k} \alpha_i}.$$

- R is an estimate for the initial residue, and G is an estimate of something like the variance of the stochastic gradient.
- For instance, for constant step size $\alpha_i = h$

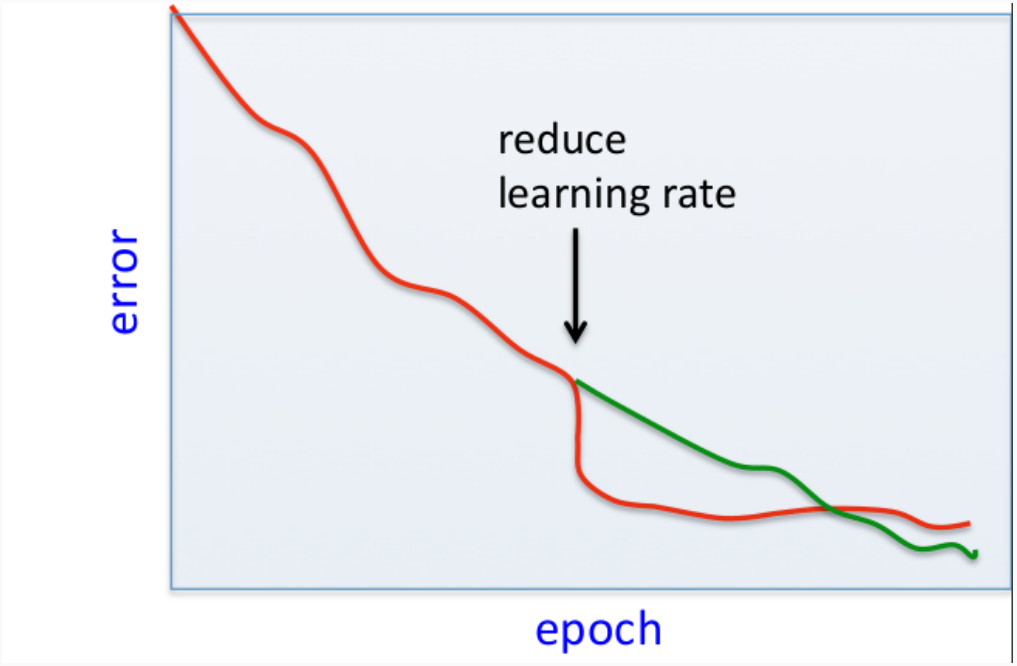
$$\mathbb{E}F(\hat{\mathbf{x}}_k) - F(\mathbf{x}_{\text{opt}}) \leq \frac{R^2}{2h(k+1)} + \frac{G^2 h}{2} \to_{k\to\infty} \frac{G^2 h}{2}.$$

Stochastic gradient descent

- SGD summary:
  - SGD comes to an "uncertainty region" of radius $\frac{1}{2}G^2h$, and it is proportional to step size;
  - the faster we go, the faster we arrive there but the larger the uncertainty region will be, i.e., we need to reduce the learning rate with time;
  - SGD converges slowly: full GD for convex functions has residue $O(1/k)$, while SGD has only $O(1/\sqrt{k})$;
  - but far from the uncertainty region we still have $O(1/k)$ residue for constant learning rate, i.e., it slows down only near the optimum, and generally our goal is to reach the uncertainty region;
  - but it still depends on G, and this will be especially important for neurobayesian approaches.

SGD WITH MOMENTUM

- We need some improvements, need to do something with the learning rate.
- Better not decrease it too quickly.



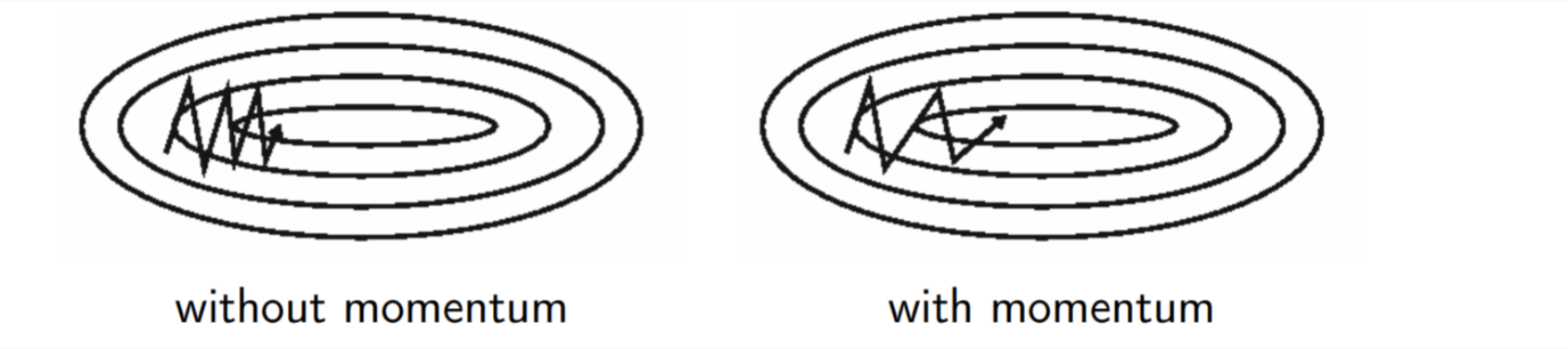
- But this does not take into account F itself, anyway.

SGD WITH MOMENTUM

- *SGD with momentum*: let's keep part of the velocity, like inertia in physics.
- We get

$$u_t = \gamma u_{t-1} + \eta \nabla_{\mathbf{x}} F(\mathbf{x}),$$
$$\mathbf{x} = \mathbf{x} - u_t.$$

- We preserve γu_{t-1} from the previous step.

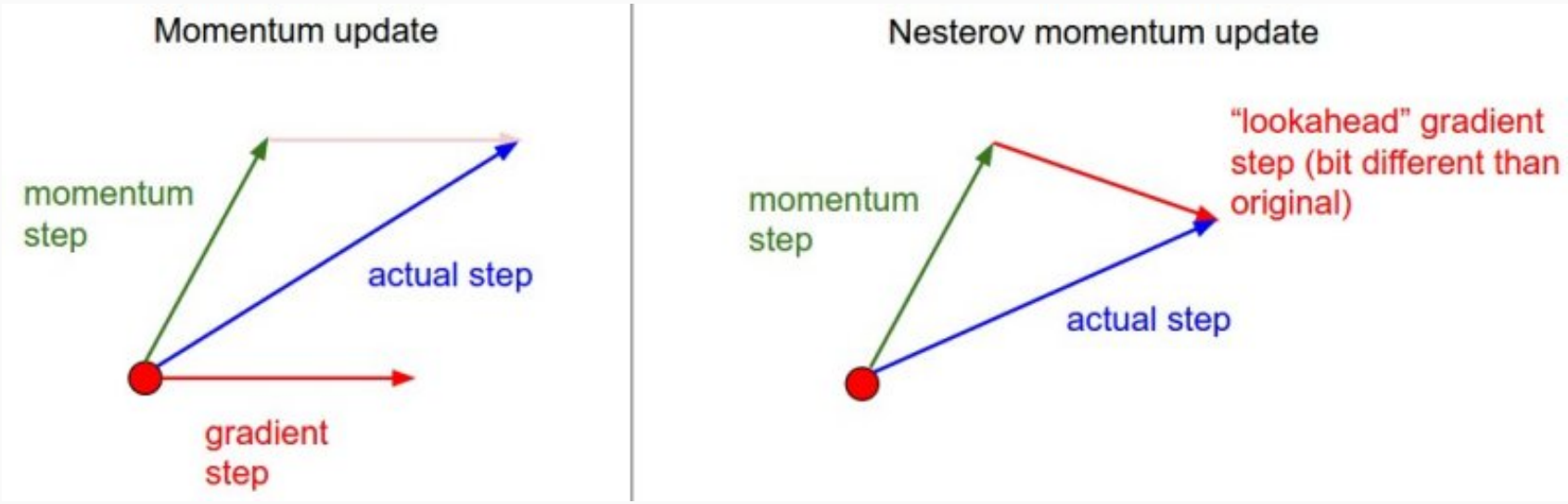


without momentum

with momentum

SGD with momentum

- But we already know we will get to the intermediate point γu_{t-1}.
- Let's compute the gradient right there!



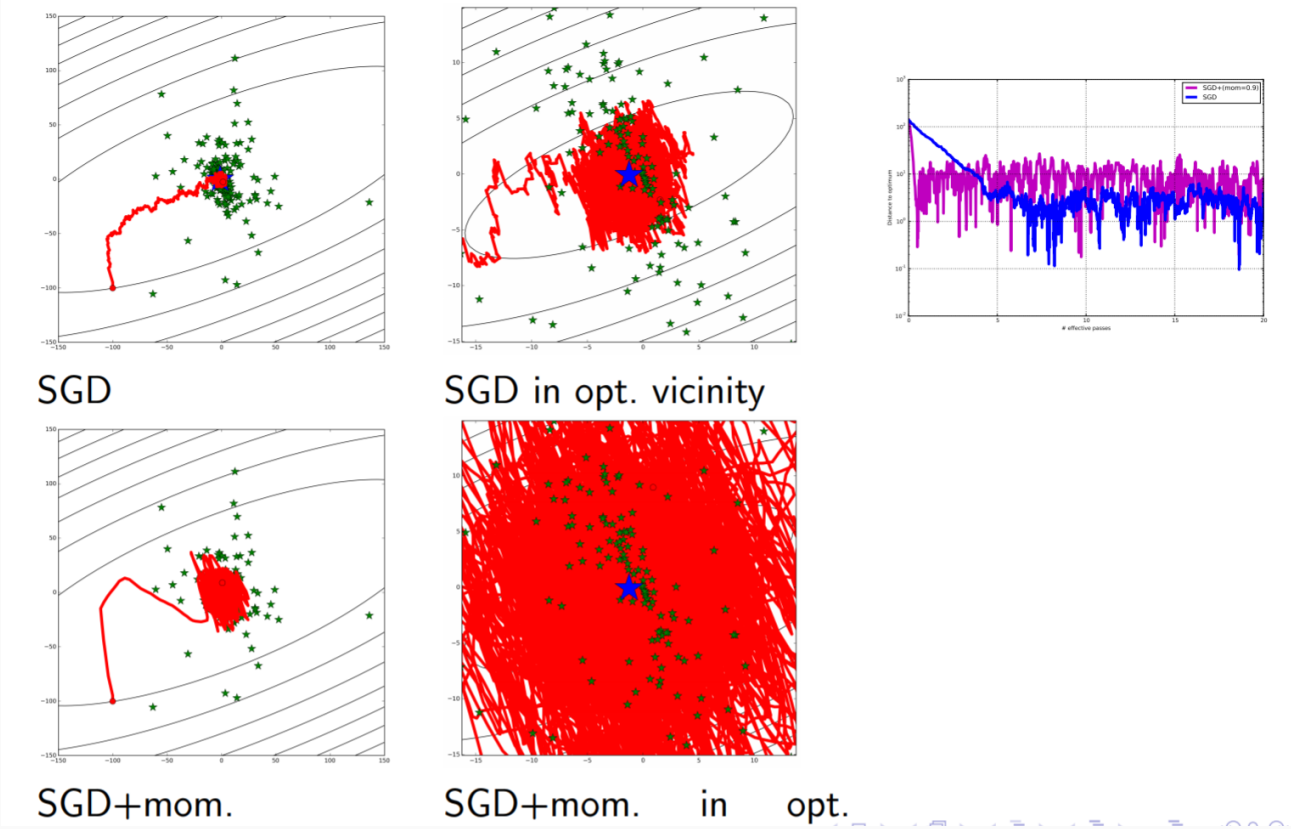
SGD WITH MOMENTUM

- *Nesterov's momentum*:

$$u_t = \gamma u_{t-1} + \eta \nabla_{\mathbf{x}} F(\mathbf{x} - \gamma u_{t-1})$$

SGD WITH MOMENTUM

- The problems are still there, of course:



- Can we do even better?..

Adaptive gradient descent

- Previously, we had the same learning rate in all directions.
- Idea: let's move faster along the axes that do not change much and slower along the faster changing parameters.

Adaptive gradient descent

- *Adagrad*: let's take into account the history of updates.
- Denoting $g_{t,i} = \nabla_{\mathbf{w}_i} L(\mathbf{w})$, we get

$$\mathbf{w}_{t+1,i} = \mathbf{w}_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i},$$

  where G_t is a diagonal matrix with $G_{t,ii} = G_{t-1,ii} + g_{t,i}^2$ that stores the total value of the gradient along the entire learning history.
- Thus, learning rate keeps decreasing but at different rates for different $\mathbf{w}_i$.
- What's the problem with Adagrad?

Adaptive gradient descent

- Problem: G keeps increasing, and the learning rate will never rebound.
- *Adadelta* (Zeiler, 2012) – same idea with two modifications.
- First, we store an exponential average of the history:

$$G_{t,ii} = \rho G_{t-1,ii} + (1-\rho) g_{t,i}^2.$$

- And then do the same kind of update:

$$\mathbf{u}_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} \mathbf{g}_{t-1}.$$

Adaptive gradient descent

- Second, we need to take “units of measurement” into account.
- There was a problem in previous approaches:
  - in regular gradient descent or with momentum the “units” of parameter updates $\Delta \mathbf{w}$ are the units of gradient, i.e., if the weights are in seconds and objective function is in meters the gradient will be measures in meters per second, and we are subtracting meters per second from seconds; they might have completely different scales;
  - in Adagrad the values of $\Delta \mathbf{w}$ resulted from ratios of gradients, so the updates were dimensionless; not so good as well.

Adaptive gradient descent

- Tnis problem does not arise in second order optimization: parameter update $\Delta\mathbf{w}$ is proportional to $H^{-1}\nabla_{\mathbf{w}} f$, i.e.,

$$\Delta\mathbf{w} \propto H^{-1}\nabla_{\mathbf{w}} f \propto \frac{\frac{\partial f}{\partial \mathbf{w}}}{\frac{\partial^2 f}{\partial \mathbf{w}^2}} \propto \text{ dimension of } \mathbf{w}.$$

- To fix *Adadelta*, let's use another exponential average, now of the squares of parameter updates.
- We approximate it with previous steps:

$$\mathbb{E}\left[\Delta\mathbf{w}^2\right]_t = \rho\mathbb{E}\left[\Delta\mathbf{w}^2\right]_{t-1} + (1-\rho)\Delta\mathbf{w}^2, \text{ where}$$

$$u_t = -\frac{\sqrt{\mathbb{E}\left[\Delta\mathbf{w}^2\right]_{t-1} + \epsilon}}{\sqrt{G_{t-1} + \epsilon}} \cdot g_{t-1}.$$

Adaptive gradient descent

- Sometimes you also hear about *RMSprop* (no real reference, Hinton's lectures).
- It's basically the same as *Adadelta*, but *RMSprop* does not use the second idea with changing measurement units:

$$u_t = -\frac{\eta}{\sqrt{G_{t-1} + \epsilon}} \cdot g_{t-1}.$$

Adaptive gradient descent

- Finally, *Adam* (Kingma, Ba, 2014) arrived.
- A modification of *Adagrad* with smoothed versions of mean and mean squared gradients:

$$
\begin{aligned}
m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t, \\
v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \\
u_t &= \frac{\eta}{\sqrt{v + \epsilon}} m_t.
\end{aligned}
$$

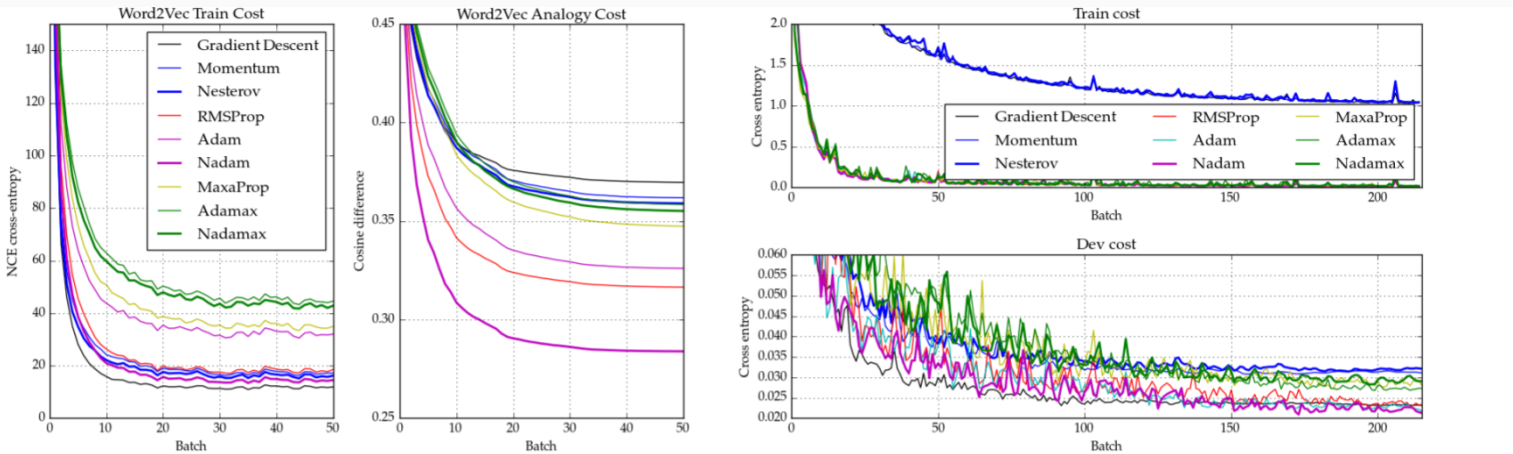
- (Kingma, Ba, 2014) recommend $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$.
- *Adam* virtually does not need tuning and is very often used in practice.
- ...[`http://ruder.io/optimizing-gradient-descent/`]...

Adaptive gradient descent

- Nadam (Dozat, 2016), a combination of Adam and Nesterov's momentum:

Algorithm 8 Nesterov-accelerated adaptive moment estimation

$$\mathbf{g}_t \leftarrow \nabla_{\theta_{t-1}} f(\theta_{t-1})$$
$$\hat{\mathbf{g}} \leftarrow \frac{\mathbf{g}_t}{1-\prod_{i=1}^{t} \mu_i}$$
$$\mathbf{m}_t \leftarrow \mu \mathbf{m}_{t-1} + (1-\mu)\mathbf{g}_t$$
$$\hat{\mathbf{m}}_t \leftarrow \frac{\mathbf{m}_t}{1-\prod_{i=1}^{t+1} \mu_i}$$
$$\mathbf{n}_t \leftarrow \nu \mathbf{n}_{t-1} + (1-\nu)\mathbf{g}_t^2$$
$$\hat{\mathbf{n}}_t \leftarrow \frac{\mathbf{n}_t}{1-\nu^t}$$
$$\bar{\mathbf{m}}_t \leftarrow (1-\mu_t)\hat{\mathbf{g}}_t + \mu_{t+1}\hat{\mathbf{m}}_t$$
$$\theta_t \leftarrow \theta_{t-1} - \eta \frac{\bar{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{n}}_t}+\varepsilon}$$



Adaptive gradient descent

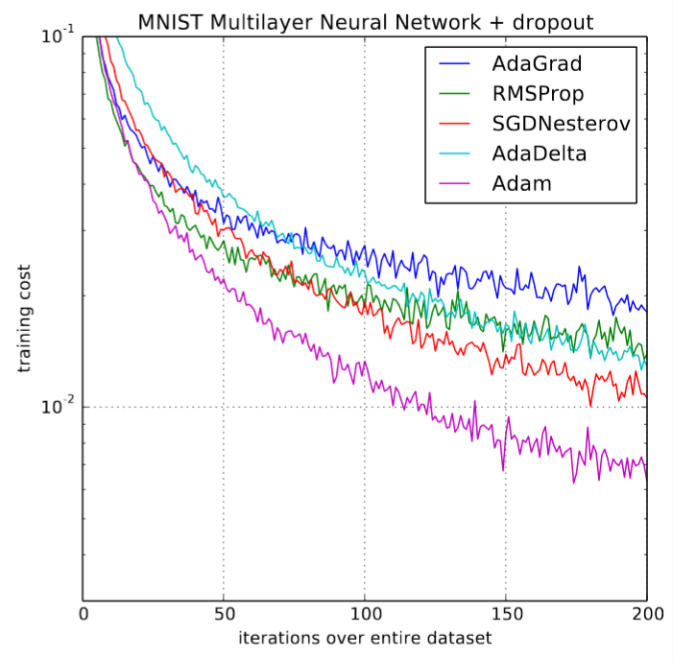
- A slightly more general view – all of this looks as follows:
  - vanilla stochastic gradient descent:
    $\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \tilde{\nabla} f(\mathbf{w}_k)$, where $\tilde{\nabla} f(\mathbf{w}_k) = \nabla f(\mathbf{w}_k; \mathbf{x}_{i_k})$;
  - SGD with momentum:
    $\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k \tilde{\nabla} f\left(\mathbf{w}_k + \gamma_k \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right)\right) + \beta_k \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right)$;
  - adaptive SGD:

    $$\mathbf{w}_{k+1} = \mathbf{w}_k - \alpha_k H_k^{-1} \tilde{\nabla} f\left(\mathbf{w}_k + \gamma_k \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right)\right) + \beta_k H_k^{-1} H_{k-1} \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right),$$

    where usually $H_k = \operatorname{diag}\left(\left[\sum_{i=1}^{k} \eta_i \mathbf{g}_i \circ \mathbf{g}_i\right]^{1/2}\right)$, where $\mathbf{g}_k = \tilde{\nabla} f\left(\mathbf{w}_k + \gamma_k \left(\mathbf{w}_k - \mathbf{w}_{k-1}\right)\right)$ (i.e., H_k is a diagonal matrix whose elements are square roots of linear combinations of squares of the previous gradients).
- Adaptive methods are trying to adapt to the data space geometry, while basic SGD uses the standard L_2-geometry with $H_k = \mathbf{I}$.

Adam, AdamW, and other animals

- When Adam appeared, it was all the buzz:



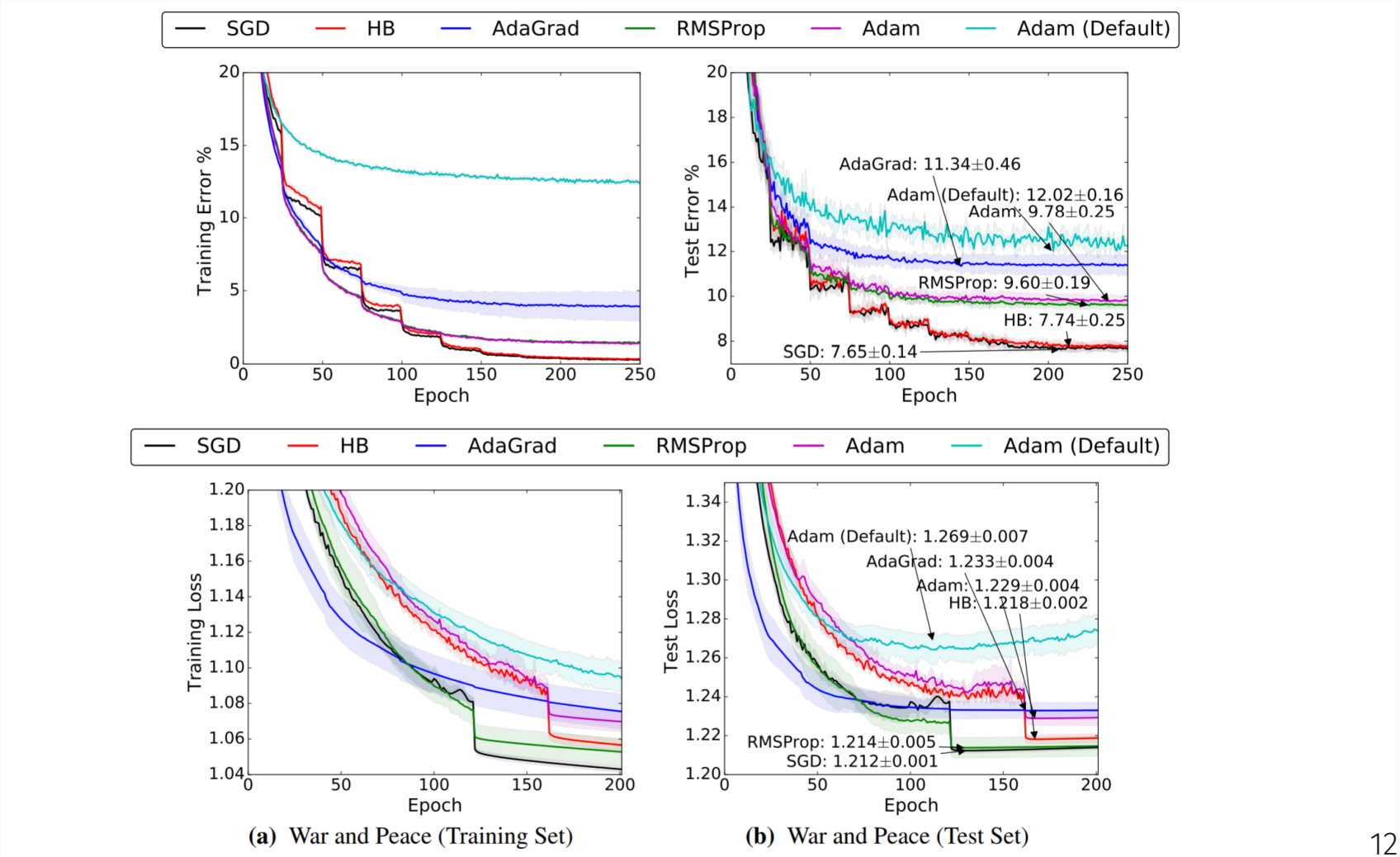
- But then people realized it's not that simple. Researchers often used plain vanilla SGD instead. Why?

Adam, AdamW, and other animals

- The Marginal Value of Adaptive Gradient Methods in Machine Learning (Wilson et al., 2017)
- Basic conclusions:
  - when the problem has several global minima, different algorithms can find different solutions from the same initial point;
  - in particular, adaptive methods can lead to overfitting, i.e., local solutions that generalize poorly;
  - and this is *not just the theoretical worst case but also happens in practice*.

Adam, AdamW, and other animals

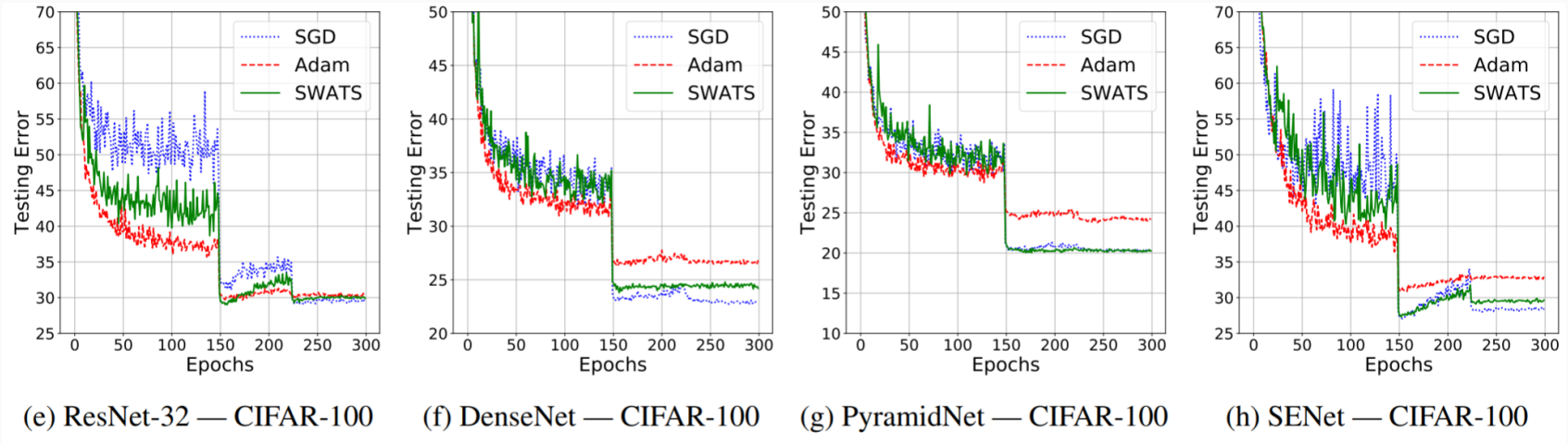
- The Marginal Value of Adaptive Gradient Methods in Machine Learning (Wilson et al., May 2017)



(a) War and Peace (Training Set) **(b)** War and Peace (Test Set)

Adam, AdamW, and other animals

- But Adam is still great at the beginning.
- That's why people suggested to, e.g., switch from Adam to SGD at the right time (Keskar, Socher, Dec 2017)



(e) ResNet-32 — CIFAR-100 (f) DenseNet — CIFAR-100 (g) PyramidNet — CIFAR-100 (h) SENet — CIFAR-100

- But this is not the whole story either...

Adam, AdamW, and other animals

- Fixing Weight Decay Regularization in Adam (Loshchilov, Hutter, Feb 2018):
  - weight decay is usually assumed to be the same as L_2-regularization;
  - but it turns out that's not quite true for adaptive methods;
  - let's dig a little deeper...

Adam, AdamW, and other animals

- Original weight decay (Hanson, Pratt, 1988):

$$\mathbf{x}_{t+1} = (1 - w)\mathbf{x}_t - \eta \nabla f_t(\mathbf{x}_t),$$

  where w is the weight decay rate, η is the learning rate.
- It is obviously equivalent to changing f:

$$f_t^{\text{reg}}(\mathbf{x}_t) = f_t(\mathbf{x}_t) + \frac{w}{2}\|\mathbf{x}_t\|_2^2.$$

- Or, easier still, changing the gradient:

$$\nabla f_t^{\text{reg}}(\mathbf{x}_t) = \nabla f_t(\mathbf{x}_t) + w\mathbf{x}_t.$$

- That's all true but...

Adam, AdamW, and other animals

- ...but doesn't work already even with momentum:

Algorithm 1 SGD with L_2 regularization and SGD with weight decay (SGDW), both with momentum

1: **given** initial learning rate $\alpha \in \mathbb{R}$, momentum factor $\beta_1 \in \mathbb{R}$, weight decay / L_2 regularization factor $w \in \mathbb{R}$
2: **initialize** time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
3: **repeat**
4: $t \leftarrow t + 1$
5: $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
6: $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w\boldsymbol{x}_{t-1}$
7: $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, be used for warm restarts
8: $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + \eta_t \alpha \boldsymbol{g}_t$
9: $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \boldsymbol{m}_t - \eta_t w \boldsymbol{x}_{t-1}$
10: **until** *stopping criterion is met*
11: **return** optimized parameters $\boldsymbol{x}_t$

- We see that $\mathbf{x}_t$ decays by $\alpha w \mathbf{x}_{t-1}$, not by $w\mathbf{x}_{t-1}$; to get the same behaviour we need to set $w_t = \frac{\alpha}{\alpha'}\delta$, and now hyperparameters α and w are connected.
- Easy solution: let's move decay to the change of $\mathbf{x}_t$ itsefl (line 9) and add scaling to η_t.

Adam, AdamW, and other animals

- The same happens with Adam:

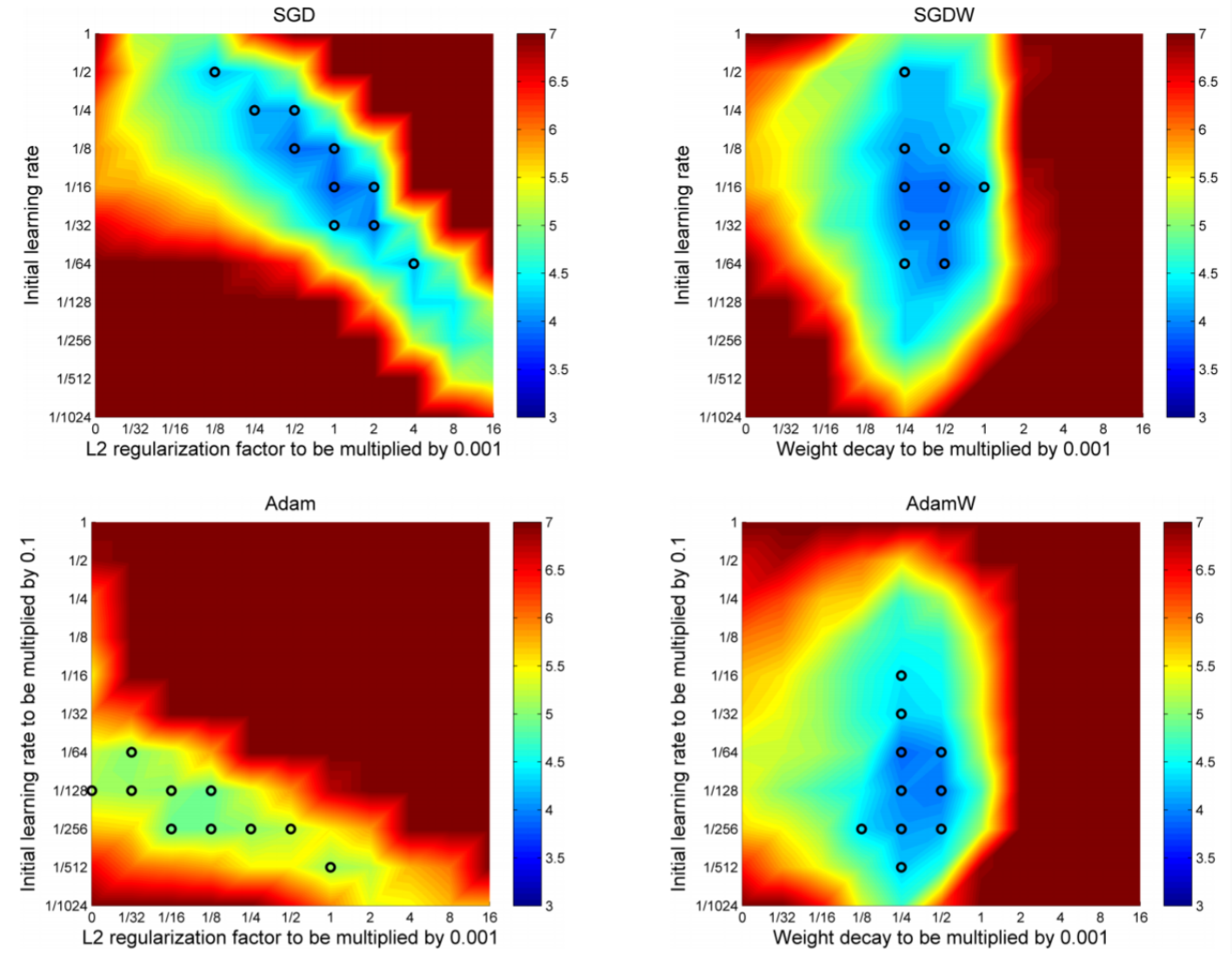
Algorithm 2 Adam with L_2 regularization and Adam with weight decay (AdamW)

given $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, w \in \mathbb{R}$
initialize time step $t \leftarrow 0$, parameter vector $\boldsymbol{x}_{t=0} \in \mathbb{R}^n$, first moment vector $\boldsymbol{m}_{t=0} \leftarrow \boldsymbol{0}$, second moment vector $\boldsymbol{v}_{t=0} \leftarrow \boldsymbol{0}$, schedule multiplier $\eta_{t=0} \in \mathbb{R}$
repeat
  $t \leftarrow t + 1$
  $\nabla f_t(\boldsymbol{x}_{t-1}) \leftarrow \text{SelectBatch}(\boldsymbol{x}_{t-1})$ ▷ select batch and return the corresponding gradient
  $\boldsymbol{g}_t \leftarrow \nabla f_t(\boldsymbol{x}_{t-1}) + w\boldsymbol{x}_{t-1}$
  $\boldsymbol{m}_t \leftarrow \beta_1 \boldsymbol{m}_{t-1} + (1 - \beta_1)\boldsymbol{g}_t$ ▷ here and below all operations are element-wise
  $\boldsymbol{v}_t \leftarrow \beta_2 \boldsymbol{v}_{t-1} + (1 - \beta_2)\boldsymbol{g}_t^2$
  $\hat{\boldsymbol{m}}_t \leftarrow \boldsymbol{m}_t / (1 - \beta_1^t)$ ▷ β_1 is taken to the power of t
  $\hat{\boldsymbol{v}}_t \leftarrow \boldsymbol{v}_t / (1 - \beta_2^t)$ ▷ β_2 is taken to the power of t
  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$ ▷ can be fixed, decay, or also be used for warm restarts
  $\boldsymbol{x}_t \leftarrow \boldsymbol{x}_{t-1} - \eta_t \left(\alpha \hat{\boldsymbol{m}}_t / (\sqrt{\hat{\boldsymbol{v}}_t} + \epsilon) + w\boldsymbol{x}_{t-1} \right)$
until *stopping criterion is met*
return optimized parameters $\boldsymbol{x}_t$

- In basic Adam, weights with large gradients decay less, which is not always a good idea.
- AdamW restores the original behaviour.

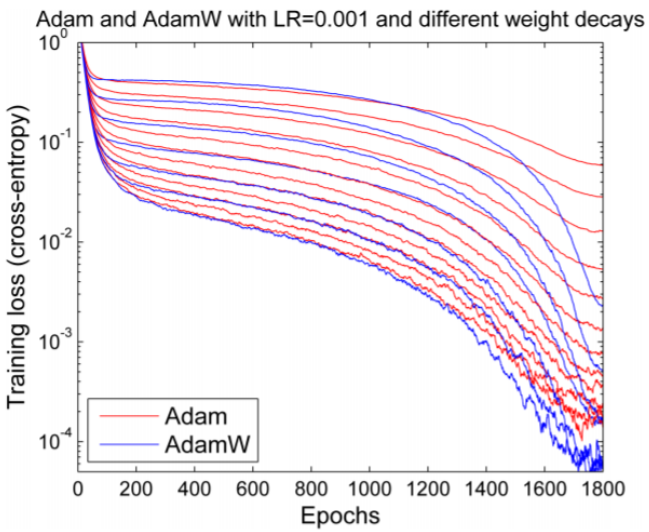
Adam, AdamW, and other animals

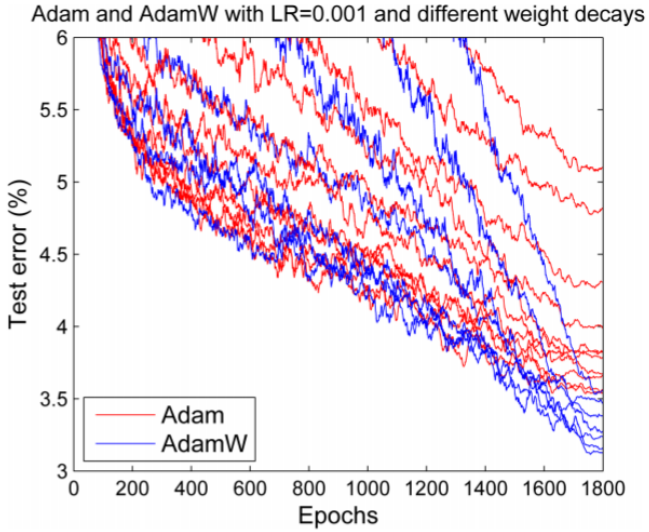
- And now the hyperparameters disentangle much better:

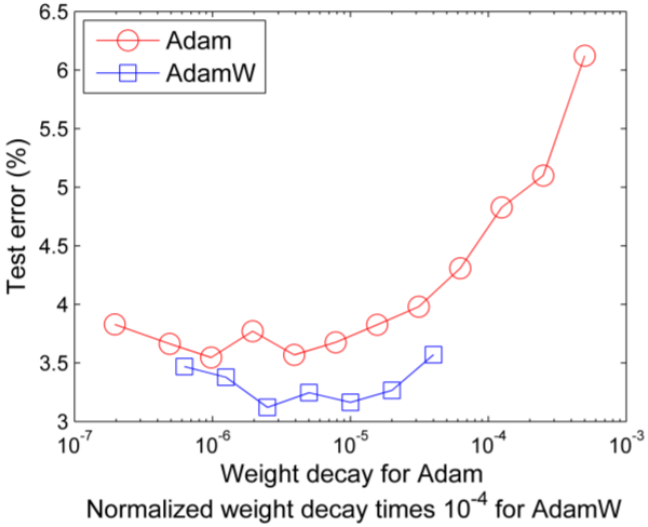


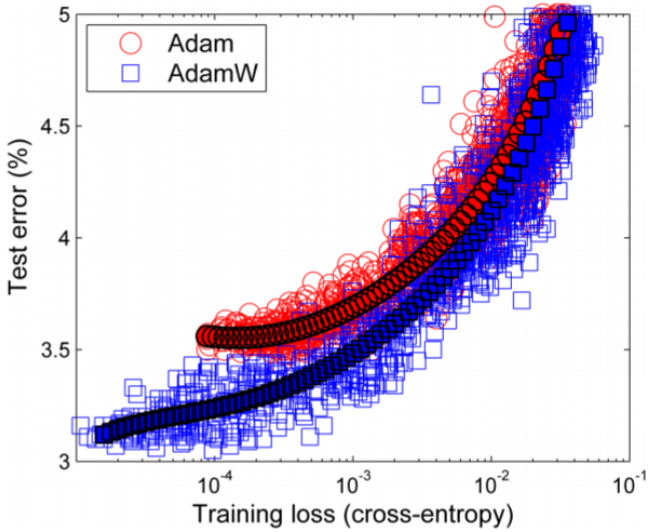
Adam, AdamW, and other animals

- And the generalization is better too:



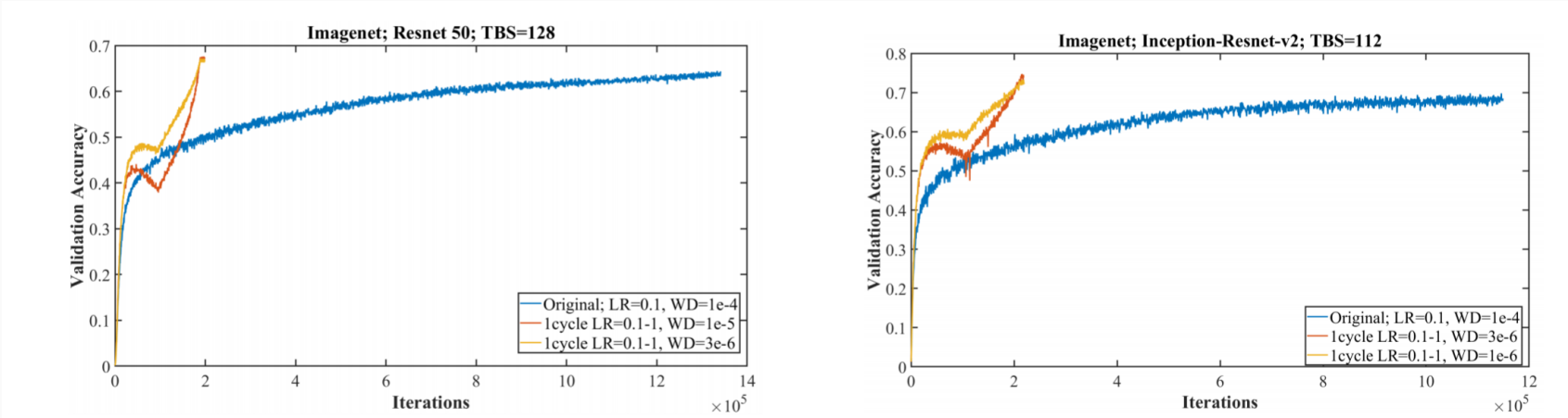






Super-Convergence and amsgrad

- Two more interesting ideas.
- Super-Convergence (Smith, Topin, 2017): let's change the learning rate cyclically.
- A combination of curriculum learning (Bengio et al., 2009) and classical simulated annealing.



- But this does not reproduce in a stable way.

Super-Convergence and amsgrad

- Another idea with a similar destiny – amsgrad (Reddi et al., 2018), ICLR 2018 best paper!
- Idea:
  - exponential average in Adam/RMSprop can hurt convergence;
  - in particular, the proof of Adam's convergence was wrong;
  - while AMSGrad keeps the maximal gradient size, and that's better.

Algorithm 2 AMSGrad

Input: $x_1 \in \mathcal{F}$, step size $\{\alpha_t\}_{t=1}^{T}$, $\{\beta_{1t}\}_{t=1}^{T}$, β_2
Set $m_0 = 0$, $v_0 = 0$ and $\hat{v}_0 = 0$
for $t = 1$ **to** T **do**
  $g_t = \nabla f_t(x_t)$
  $m_t = \beta_{1t} m_{t-1} + (1 - \beta_{1t}) g_t$
  $v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$
  $\hat{v}_t = \max(\hat{v}_{t-1}, v_t)$ and $\hat{V}_t = \mathrm{diag}(\hat{v}_t)$
  $x_{t+1} = \Pi_{\mathcal{F}, \sqrt{\hat{V}_t}}(x_t - \alpha_t m_t / \sqrt{\hat{v}_t})$
end for

- But this is not really confirmed in practice either...

Thank you!

Thank you for your attention!