DEEP LEARNING

Sergey Nikolenko

Harbour Space University, Barcelona, Spain
April 13, 2017

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

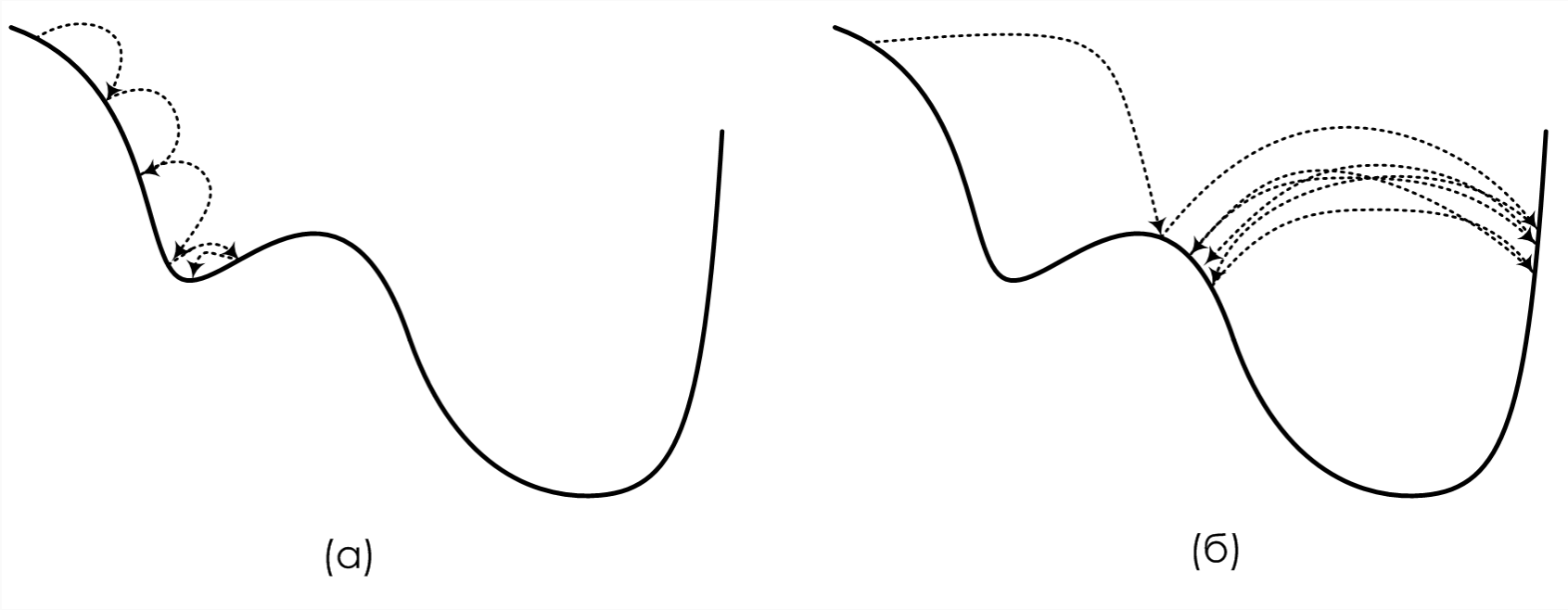
- Hence, *stochastic gradient descent*: after every training sample update

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

- 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

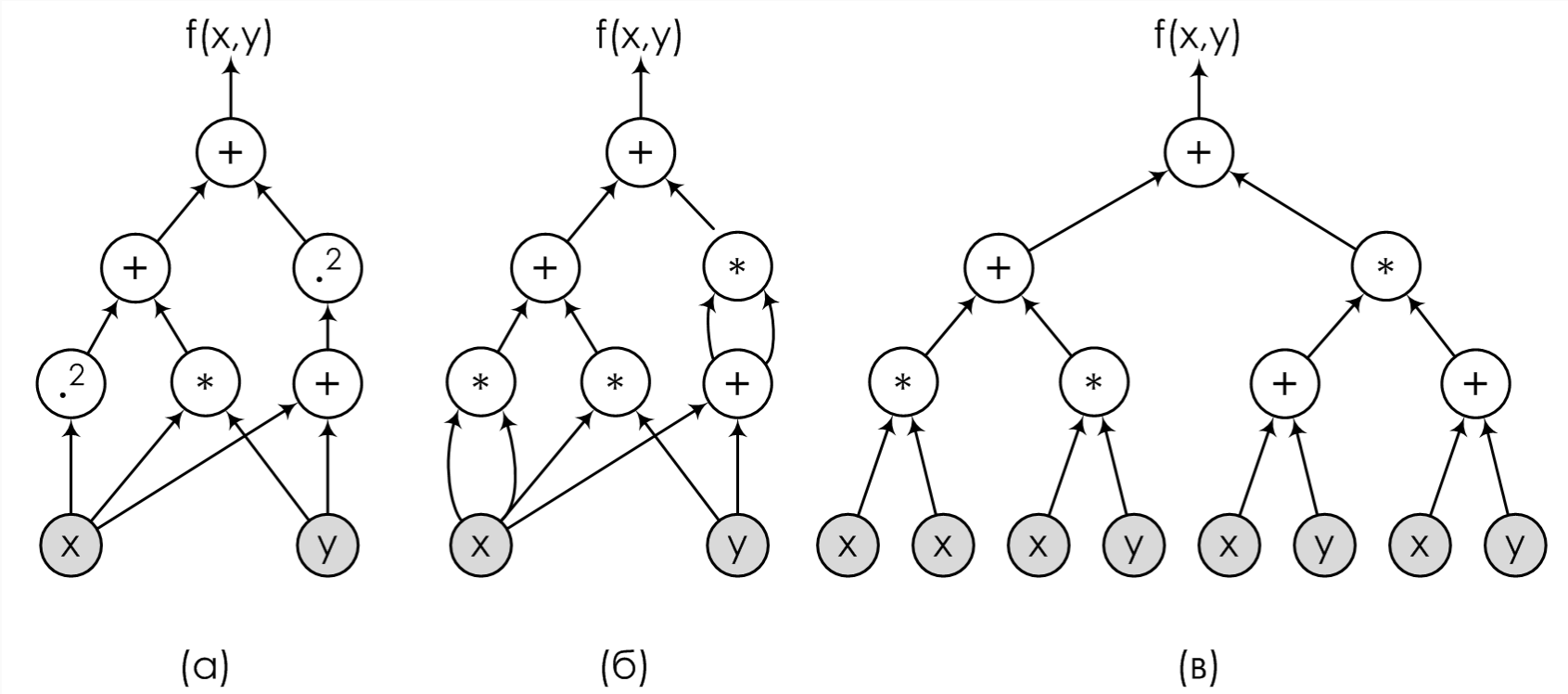
- Lots of problems with η:



- 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, FROP 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:

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

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

- 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:

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

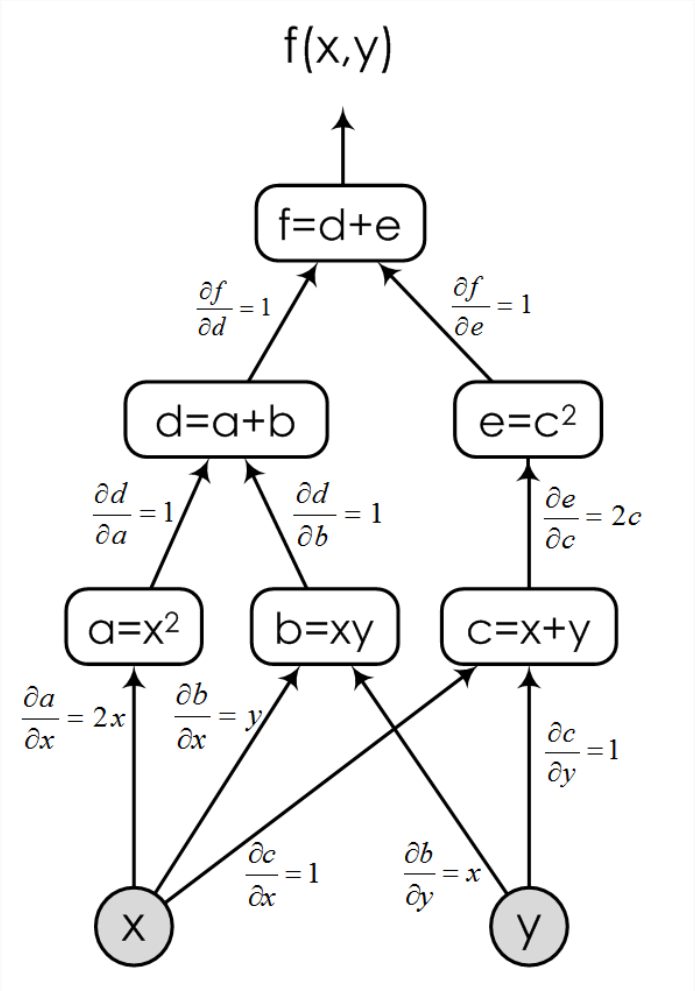
$$\nabla_{\mathbf{x}} f = \frac{\partial f}{\partial g_1}\nabla_{\mathbf{x}} g_1 + \dots + \frac{\partial f}{\partial g_k}\nabla_{\mathbf{x}} g_k = \sum_{i=1}^{k} \frac{\partial f}{\partial g_i}\nabla_{\mathbf{x}} g_i.$$

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

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

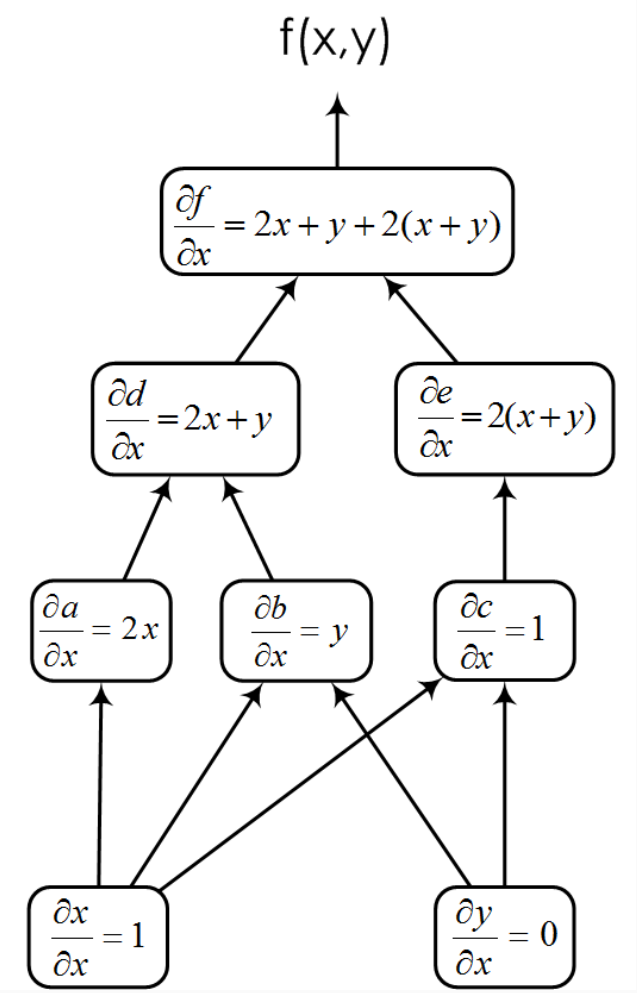
COMPUTATIONAL GRAPH, FROP AND BPROP

- Let's now go back to the example:



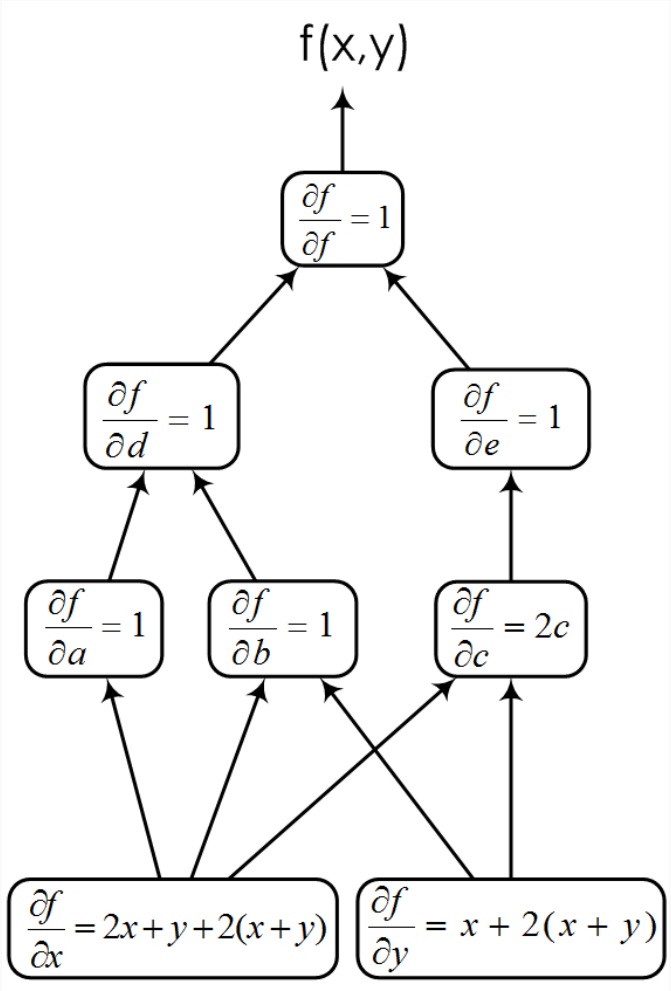
COMPUTATIONAL GRAPH, FROP AND BPROP

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



COMPUTATIONAL GRAPH, FROP 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, FROP AND BPROP

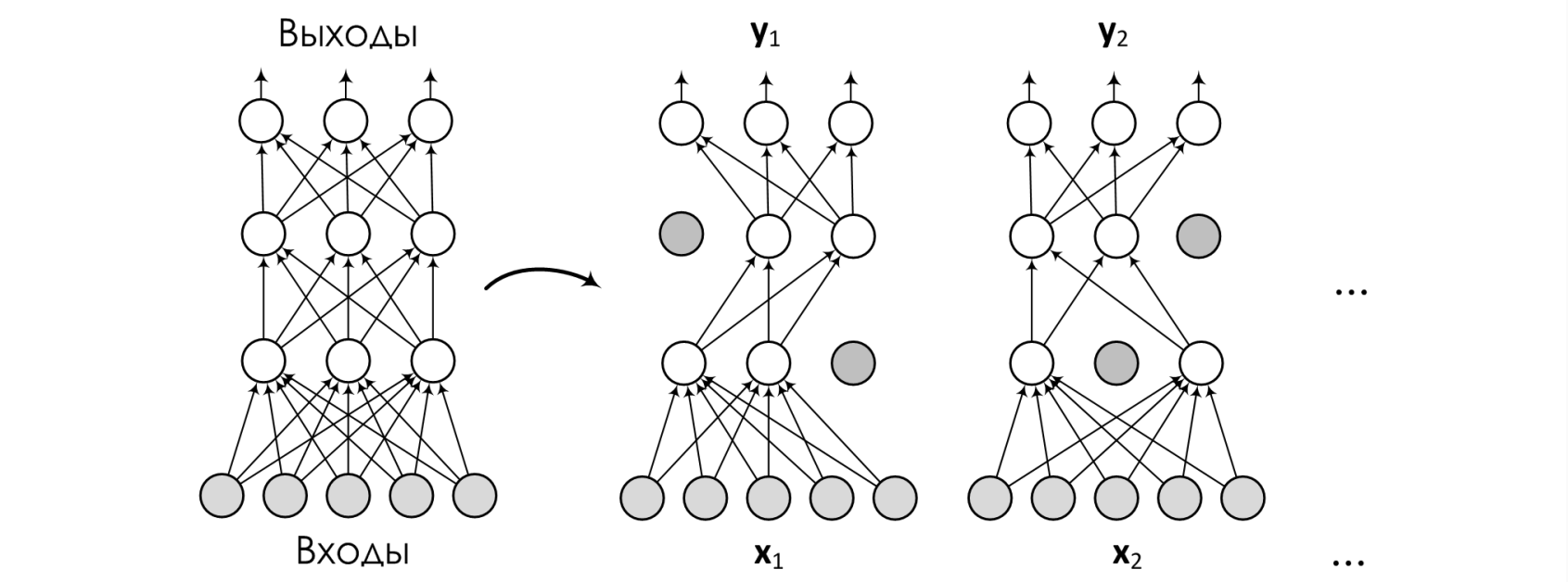
- Backprop is much better: we get all derivatives in a single pass through the graph.
- Aaaand... that's it! We can now take the gradients of any complicated composition of simple functions.
- Which is all we need to apply gradient descent!
- The libraries – *theano*, *TensorFlow* – are actually *automatic 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 "algorithms" to compute the value and the derivative.

REGULARIZATION IN NEURAL NETWORKS

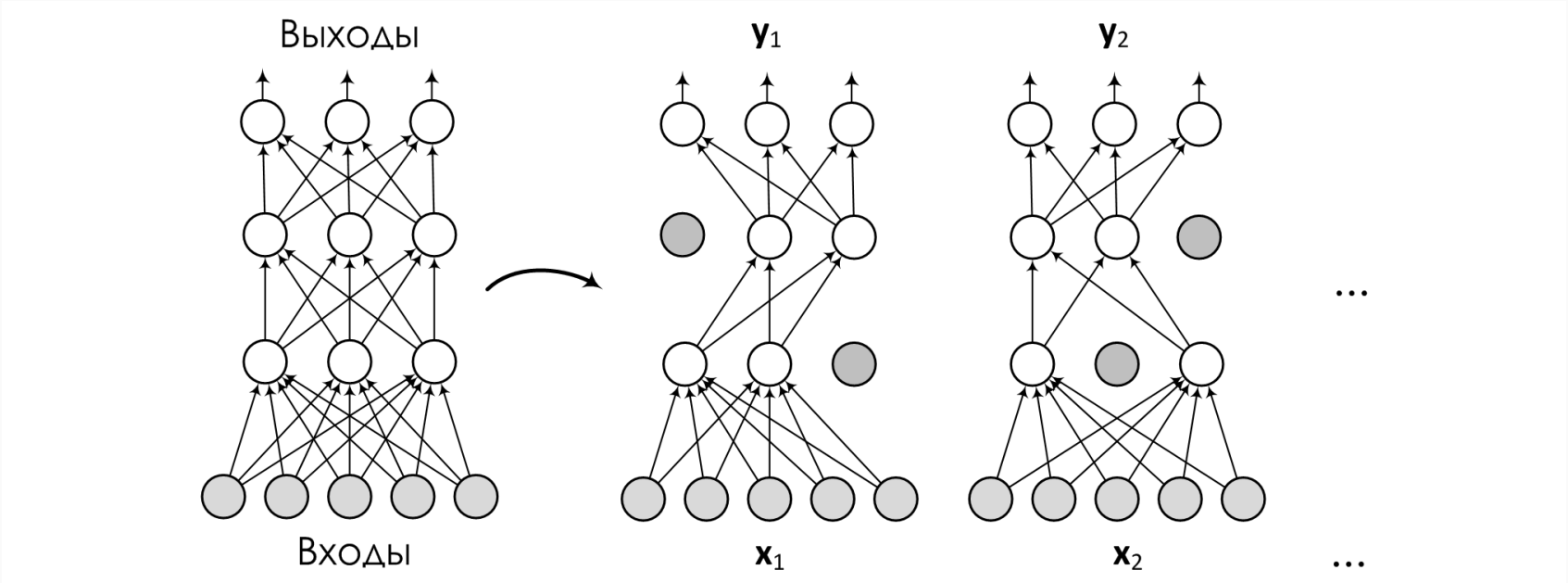
Regularization in neural networks

- NNs have *lots* of parameters.
- Regularization is necessary.
- L_2 or L_1 regularization ($\lambda \sum_w w^2$ or $\lambda \sum_w |w|$) is called *weight decay*.
- Very easy to add, just another term in the objective function.
- Sometimes still useful.

- But there are better ways.
- *Dropout*: remove some units at random with probability p!



- To apply, simply multiply the result by $1/p$ (preserving average output); and you can usually take $p = \frac{1}{2}$.



REGULARIZATION IN NEURAL NETWORKS

- Dropout improved everything *drastically*. What the... why does it work?
- Idea 1: we are making the units learn features by themselves, without relying on the others.
- Idea 2: we are kind of *averaging* a huge number of networks with shared weights, training each for one step. Like bootstrapping taken to the extreme.
- Idea 3: this is just like sex!
- Idea 4: dropout is a special kind of prior (this has led to proper dropout in recurrent NNs).

WEIGHT INITIALIZATION

WEIGHT INITIALIZATION

- The deep learning revolution began with *unsupervised pretraining*.
- Main idea: get to a good region of the search space, then fine-tune with gradient descent.
- Turns out by now we don't need unsupervised pretraining with complex models like RBM to get to a good region.
- Weight initialization is an important part of why.

WEIGHT INITIALIZATION

- *Xavier initialization* (Glorot, Bengio, 2010).
- Let's consider a single linear unit:

$$y = \mathbf{w}^\top \mathbf{x} + b = \sum_i w_i x_i + b.$$

- The variance is

$$\begin{aligned}\operatorname{Var}[y_i] = \operatorname{Var}[w_i x_i] = \mathbb{E}\left[X^2 Y^2\right] - \left(\mathbb{E}[XY]\right)^2 = \\ = \mathbb{E}[x_i]^2 \operatorname{Var}[w_i] + \mathbb{E}[w_i]^2 \operatorname{Var}[x_i] + \operatorname{Var}[w_i] \operatorname{Var}[x_i].\end{aligned}$$

WEIGHT INITIALIZATION

- The variance is

$$\begin{aligned}\operatorname{Var}[y_i] = \operatorname{Var}[w_i x_i] = \mathbb{E}\left[X^2 Y^2\right] - \left(\mathbb{E}[XY]\right)^2 = \\ = \mathbb{E}[x_i]^2 \operatorname{Var}[w_i] + \mathbb{E}[w_i]^2 \operatorname{Var}[x_i] + \operatorname{Var}[w_i] \operatorname{Var}[x_i] .\end{aligned}$$

- For symmetric activation functions and zero mean of the weights

$$\operatorname{Var}[y_i] = \operatorname{Var}[w_i] \operatorname{Var}[x_i] .$$

- And if w_i and x_i are initialized independently from the same distribution,

$$\operatorname{Var}[y] = \operatorname{Var}\left[\sum_{i=1}^{n_{\text{out}}} y_i\right] = \sum_{i=1}^{n_{\text{out}}} \operatorname{Var}[w_i x_i] = n_{\text{out}} \operatorname{Var}[w_i] \operatorname{Var}[x_i] .$$

- In other words, the output variance is proportional to the input variance with coefficient $n_{\text{out}} \operatorname{Var}[w_i]$.

WEIGHT INITIALIZATION

- Before (Glorot, Bengio, 2010), the standard way to initialize was (it's all over older literature)

$$w_i \sim U\left[-\frac{1}{\sqrt{n_{\text{out}}}}, \frac{1}{\sqrt{n_{\text{out}}}}\right].$$

- So in this case we get

$$\text{Var}\left[w_i\right] = \frac{1}{12}\left(\frac{1}{\sqrt{n_{\text{out}}}} + \frac{1}{\sqrt{n_{\text{out}}}}\right)^2 = \frac{1}{3n_{\text{out}}}, \text{ so}$$

$$n_{\text{out}}\text{Var}\left[w_i\right] = \frac{1}{3},$$

and after a few layers the signal dies down; the same happens in backprop.

WEIGHT INITIALIZATION

- Xavier initialization tries to reduce the change in variance, so we take

$$\mathrm{Var}\left[w_i\right] = \frac{2}{n_{\mathrm{in}} + n_{\mathrm{out}}},$$

  which for uniform distribution means

$$w_i \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{\mathrm{in}} + n_{\mathrm{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\mathrm{in}} + n_{\mathrm{out}}}}\right].$$

- But it only works for symmetric activations, i.e., not for ReLU...

WEIGHT INITIALIZATION

- ...until (He et al., 2015)! Let's go back to

$$\operatorname{Var}[w_i x_i] = \mathbb{E}[x_i]^2 \operatorname{Var}[w_i] + \mathbb{E}[w_i]^2 \operatorname{Var}[x_i] + \operatorname{Var}[w_i] \operatorname{Var}[x_i]$$

- We now can only make the second term zero:

$$\operatorname{Var}[w_i x_i] = \mathbb{E}[x_i]^2 \operatorname{Var}[w_i] + \operatorname{Var}[w_i] \operatorname{Var}[x_i] = \operatorname{Var}[w_i] \mathbb{E}[x_i^2], \text{ so}$$

$$\operatorname{Var}\left[y^{(l)}\right] = n_{\text{in}}^{(l)} \operatorname{Var}\left[w^{(l)}\right] \mathbb{E}\left[\left(x^{(l)}\right)^2\right].$$

- We now can only make the second term zero:

$$\mathrm{Var}\left[y^{(l)}\right] = n_{\mathrm{in}}^{(l)} \mathrm{Var}\left[w^{(l)}\right] \mathbb{E}\left[\left(x^{(l)}\right)^2\right].$$

- Suppose now that $x^{(l)} = \max(0, y^{(l-1)})$, and $y^{(l-1)}$ has a symmetric distribution around zero. Then

$$\mathbb{E}\left[\left(x^{(l)}\right)^2\right] = \frac{1}{2}\mathrm{Var}\left[y^{(l-1)}\right], \quad \mathrm{Var}\left[y^{(l)}\right] = \frac{n_{\mathrm{in}}^{(l)}}{2}\mathrm{Var}\left[w^{(l)}\right]\mathrm{Var}\left[y^{(l-1)}\right].$$

- And this leads to the variance for ReLU init; there is no n_{out} now:

$$\mathrm{Var}\left[w_i\right] = 2/n_{\mathrm{in}}^{(l)}.$$

- You don't have to make it uniform, btw; e.g., a normal distribution is fine:

$$w_i \sim \mathcal{N}\left(0, \sqrt{2/n_{\mathrm{in}}^{(l)}}\right).$$

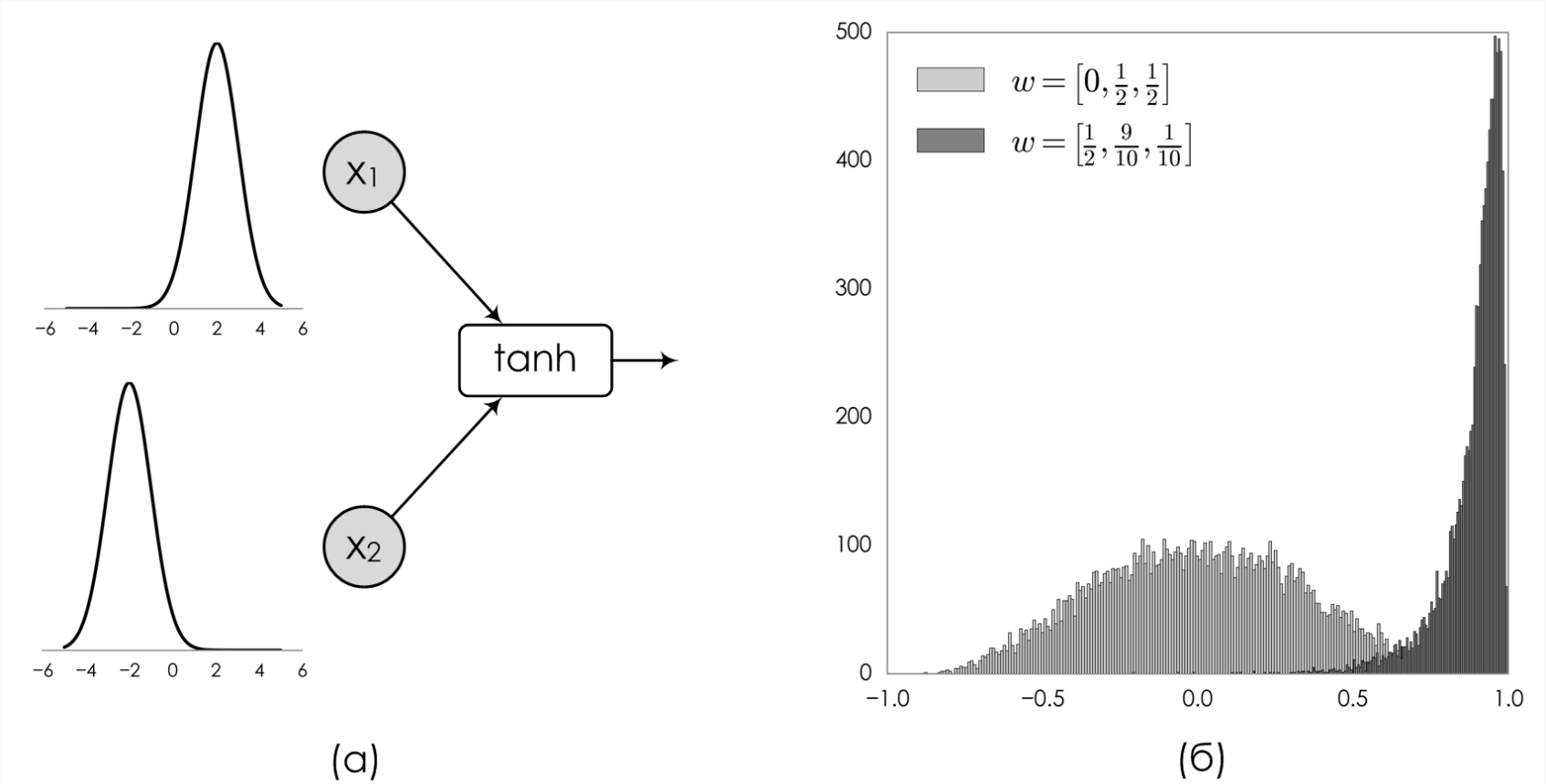
BATCH NORMALIZATION

BATCH NORMALIZATION

- Important problem in deep neural networks: *internal covariate shift*.
- When we change the weights of a layer, the distribution of its outputs changes.
- This means that the next layer has to re-train almost from scratch, it did not expect these outputs.
- Moreover, these neurons might have already reached saturation, so they can't re-train quickly.
- This seriously impedes learning.

BATCH NORMALIZATION

- A characteristic example; note how different the distributions are:



- What can we do?

BATCH NORMALIZATION

- We could try to normalize (whiten) after every layer.
- Does not work: consider a layer that simply adds a bias b to its inputs u:

$$\hat{\mathbf{x}} = \mathbf{x} - \mathbb{E}[\mathbf{x}], \text{ where } \mathbf{x} = u + b.$$

- On the next gradient descent step, we'll have $b := b + \Delta b$...
- ...but $\hat{\mathbf{x}}$ will not change:

$$u + b + \Delta b - \mathbb{E}[u + b + \Delta b] = u + b - \mathbb{E}[u + b].$$

- So the biases will simply increase unboundedly, and that's all the training we'll get; not a good thing.

BATCH NORMALIZATION

- We can try to add normalization as a layer:

$$\hat{\mathbf{x}} = \mathrm{Norm}(\mathbf{x}, \mathcal{X}).$$

- But note that the entire dataset $\mathcal{X}$ is required here.
- So on the gradient descent step we'll need to compute $\frac{\partial \mathrm{Norm}}{\partial \mathbf{x}}$ and $\frac{\partial \mathrm{Norm}}{\partial \mathcal{X}}$, and also the covariance matrix

$$\mathrm{Cov}[\mathbf{x}] = \mathbb{E}_{\mathbf{x} \in \mathcal{X}} \left[\mathbf{x}\mathbf{x}^\top\right] - \mathbb{E}\left[\mathbf{x}\right] \mathbb{E}\left[\mathbf{x}\right]^\top.$$

- Definitely won't work.

BATCH NORMALIZATION

- The solution is to normalize each component separately, and not over the whole dataset but over the current mini-batch; hence *batch normalization*.
- After batch normalization we get

$$\widehat{x}_k = \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\mathrm{Var}[x_k]}},$$

  where the statistics are computed over the current mini-batch.
- However, one more problem: now nonlinearities disappear!
- E.g., we will almost always get into the region where σ is very close to linear.

BATCH NORMALIZATION

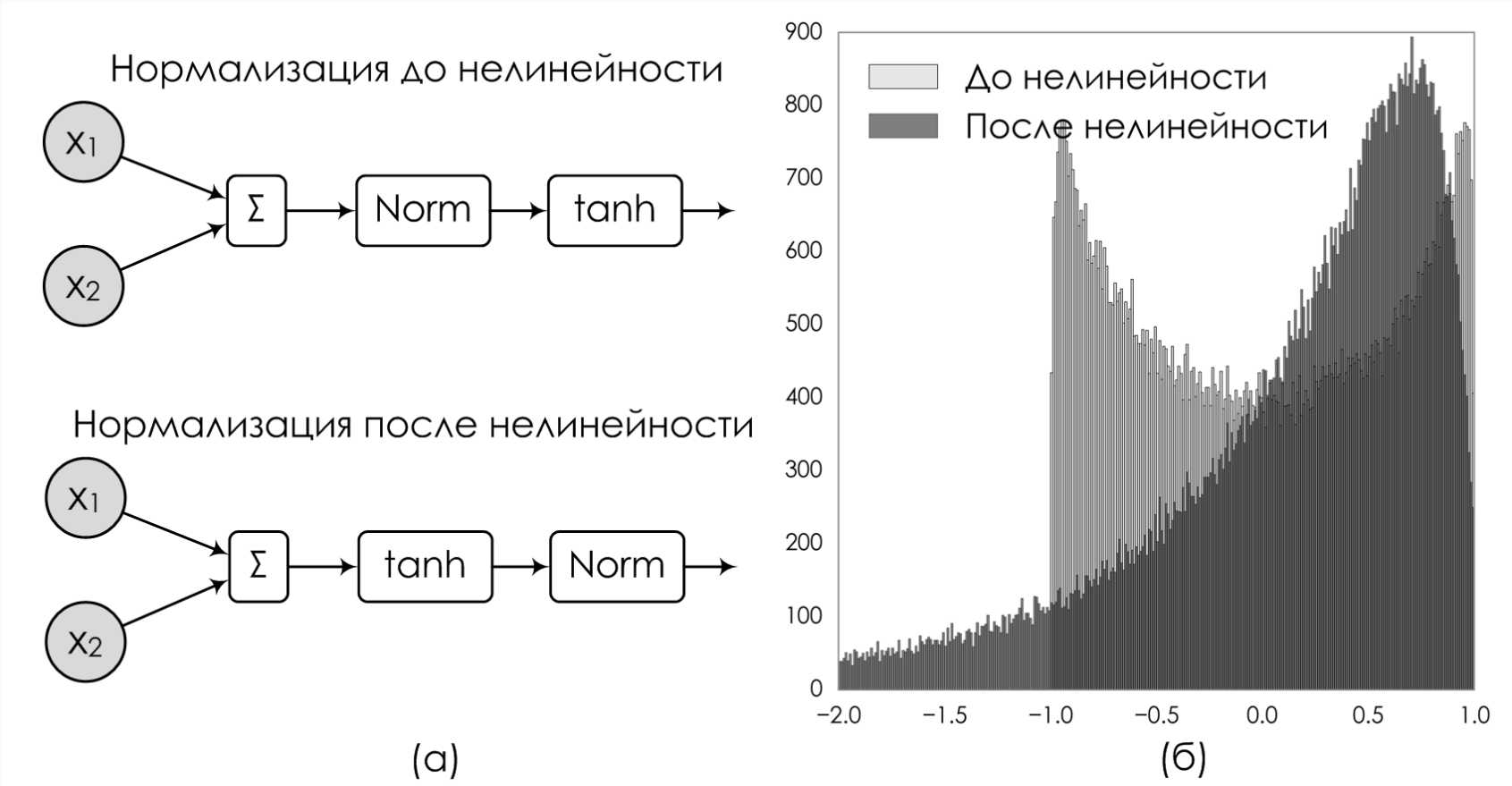
- To fix this, we have to allow the batchnorm layer enough flexibility to sometimes do *nothing* with the inputs.
- So we introduce additional shift and scale parameters:

$$y_k = \gamma_k \hat{x}_k + \beta_k = \gamma_k \frac{x_k - \mathbb{E}[x_k]}{\sqrt{\mathrm{Var}[x_k]}} + \beta_k.$$

- γ_k and β_k are new variables and will be trained just like the weights.

BATCH NORMALIZATION

- Last remark: it matters where to put the batchnorm.
- You can put it either before or after the nonlinearity.



VARIATIONS OF GRADIENT DESCENT

MOMENTUM

- Gradient descent:

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

- It all depends on the learning rate η.
- First idea – let's make it decrease over time:
  - linear decay:

$$\eta = \eta_0 \left(1 - \frac{t}{T}\right);$$

  - exponential decay:

$$\eta = \eta_0 e^{-\frac{t}{T}}.$$

- But this does not take E into account; it's better to be *adaptive*.

MOMENTUM

- *Momentum methods*: let's keep part of the speed, like a real material point would.
- With the inertia we now have

$$u_t = \gamma u_{t-1} + \eta \nabla_\theta E(\theta),$$
$$\theta = \theta - u_t.$$

- So we now preserve γu_{t-1}.

MOMENTUM

- But we already know we will go to γu_{t-1}!
- Why don't we compute the gradients right there, halfway?
- *Nesterov's momentum*:

$$u_t = \gamma u_{t-1} + \eta \nabla_\theta E(\theta - \gamma u_{t-1})$$

- Can we do even better?..

MOMENTUM

- ...well, yeah, we can try second-order methods.
- Newton's method:

$$E(\theta) \approx E(\theta_0) + \nabla_\theta E(\theta_0)(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)^\top H(E(\theta))(\theta - \theta_0).$$

- This is usually much faster, and there's nothing to tune (no η).
- But we need to compute the Hessian $H(E(\theta))$, and this is infeasible.
- Interesting problem: can we make Newton's method work for deep learning?

- But we can still do better!
- Note that so far the learning rate was the same in all directions.
- Idea: rate of change should be higher for parameters that do not change much over the input samples, and lower for highly variable parameters.
- Denoting $g_{t,i} = \nabla_{\theta_i} L(\theta)$, we get

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

  where G_t is a diagonal matrix with $G_{t,ii} = G_{t-1,ii} + g_{t,i}^2$ that accumulates the total gradient value over learning history.
- So learning rate always goes down, but at different rates for different θ_i.

- One problem: G keeps increasing, and learning rate sometimes decreases too rapidly.
- *Adadelta* – same idea, but gradient history is computed with decay:

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

- The rest is the same:

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

THANK YOU!

Thank you for your attention!