Deep Learning

Lecture 4: Training neural networks

Prof. Gilles Louppe
g.louppe@uliege.be

LIÈGE université

Today

How to optimize parameters efficiently?

- Optimizers
- Initialization
- Normalization

Optimizers

Gradient descent

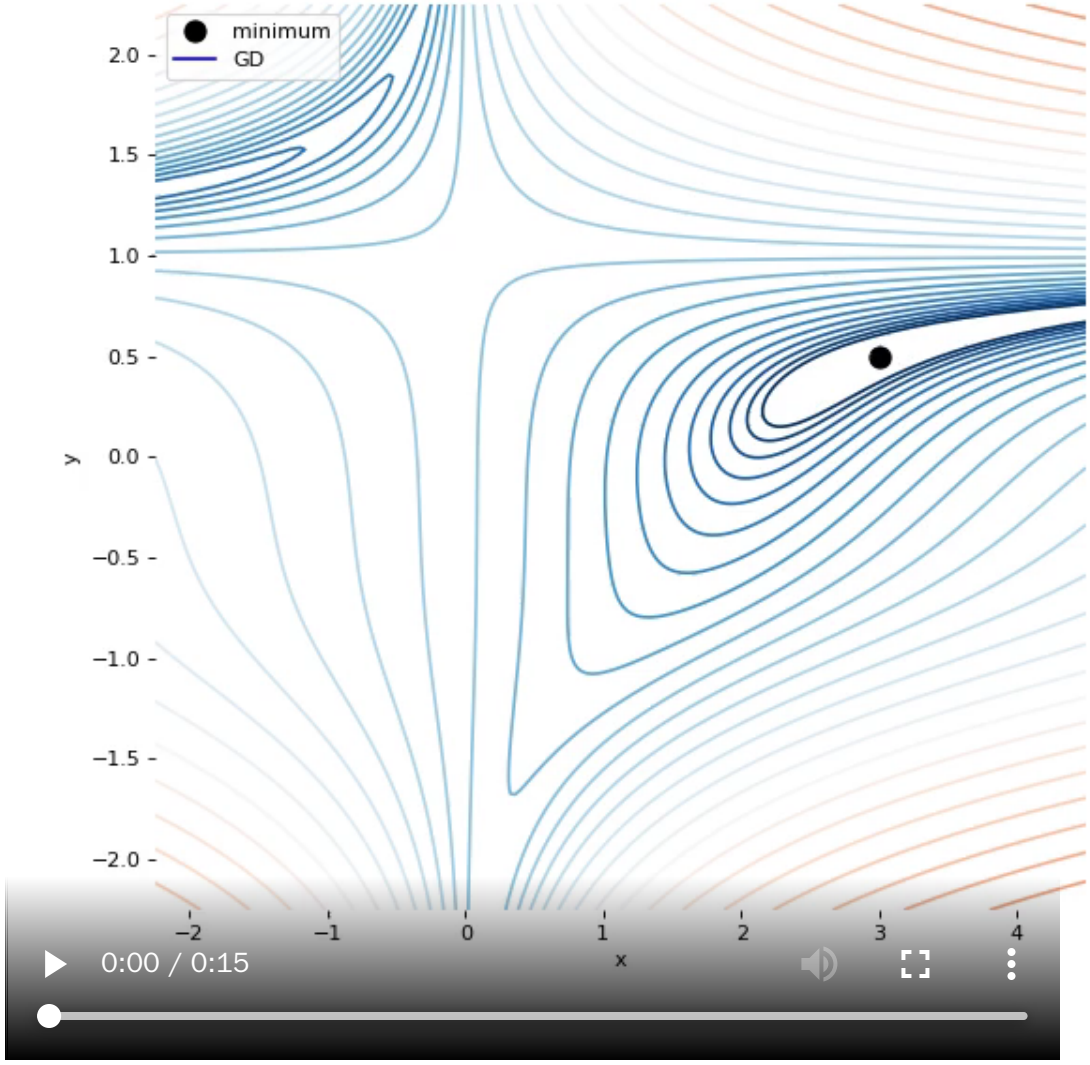
To minimize a loss $\mathcal{L}(\theta)$ of the form

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(\mathbf{x}_n; \theta)),$$

standard batch gradient descent (GD) consists in applying the update rule

$$g_t = \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \ell(y_n, f(\mathbf{x}_n; \theta_t))$$
$$\theta_{t+1} = \theta_t - \gamma g_t,$$

where γ is the learning rate.


minimum
GD
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2
−1
0
1
2
3
4
x
0:00 / 0:15

While it makes sense in principle to compute the gradient exactly,

- it takes time to compute and becomes inefficient for large N,
- it is an empirical estimation of an hidden quantity (the expected risk), and any partial sum is also an unbiased estimate, although of greater variance.

To illustrate how partial sums are good estimates, consider an ideal case where the training set is the same set of $M \ll N$ samples replicated K times. Then,

$$\begin{aligned}\mathcal{L}(\theta) &= \frac{1}{N}\sum_{i=n}^{N} \ell(y_n, f(\mathbf{x}_n; \theta)) \\ &= \frac{1}{N}\sum_{k=1}^{K}\sum_{m=1}^{M} \ell(y_m, f(\mathbf{x}_m; \theta)) \\ &= \frac{1}{N} K \sum_{m=1}^{M} \ell(y_m, f(\mathbf{x}_m; \theta)).\end{aligned}$$

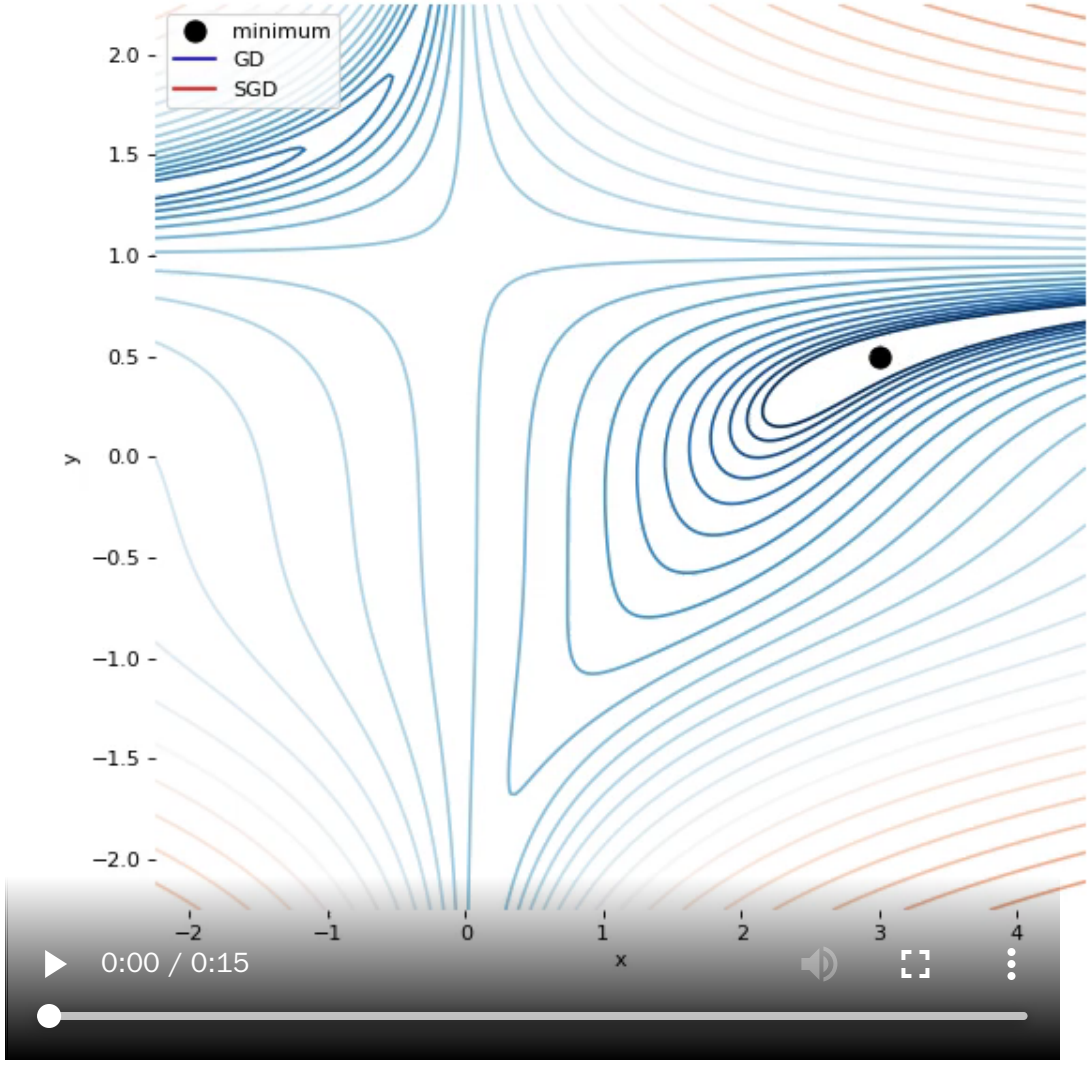
Then, instead of summing over all the samples and moving by γ, we can visit only $M = N/K$ samples and move by $K\gamma$, which would cut the computation by K.

Although this is an ideal case, there is redundancy in practice that results in similar behaviors.

Stochastic gradient descent

To reduce the computational complexity, stochastic gradient descent (SGD) consists in updating the parameters after every sample

$$
\begin{aligned}
g_t &= \nabla_\theta \ell(y_{n(t)}, f(\mathbf{x}_{n(t)}; \theta_t)) \\
\theta_{t+1} &= \theta_t - \gamma g_t.
\end{aligned}
$$


minimum
GD
SGD
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2
−1
0
1
2
3
4
x
0:00 / 0:15

While being faster than batch gradient descent,

- gradient estimates used by SGD can be very noisy,
- SGD does not benefit from the speed-up of batch-processing.

Mini-batching

Instead, mini-batch SGD consists of visiting the samples in mini-batches and updating the parameters each time

$$g_t = \frac{1}{B} \sum_{b=1}^{B} \nabla_\theta \ell(y_{n(t,b)}, f(\mathbf{x}_{n(t,b)}; \theta_t))$$
$$\theta_{t+1} = \theta_t - \gamma g_t,$$

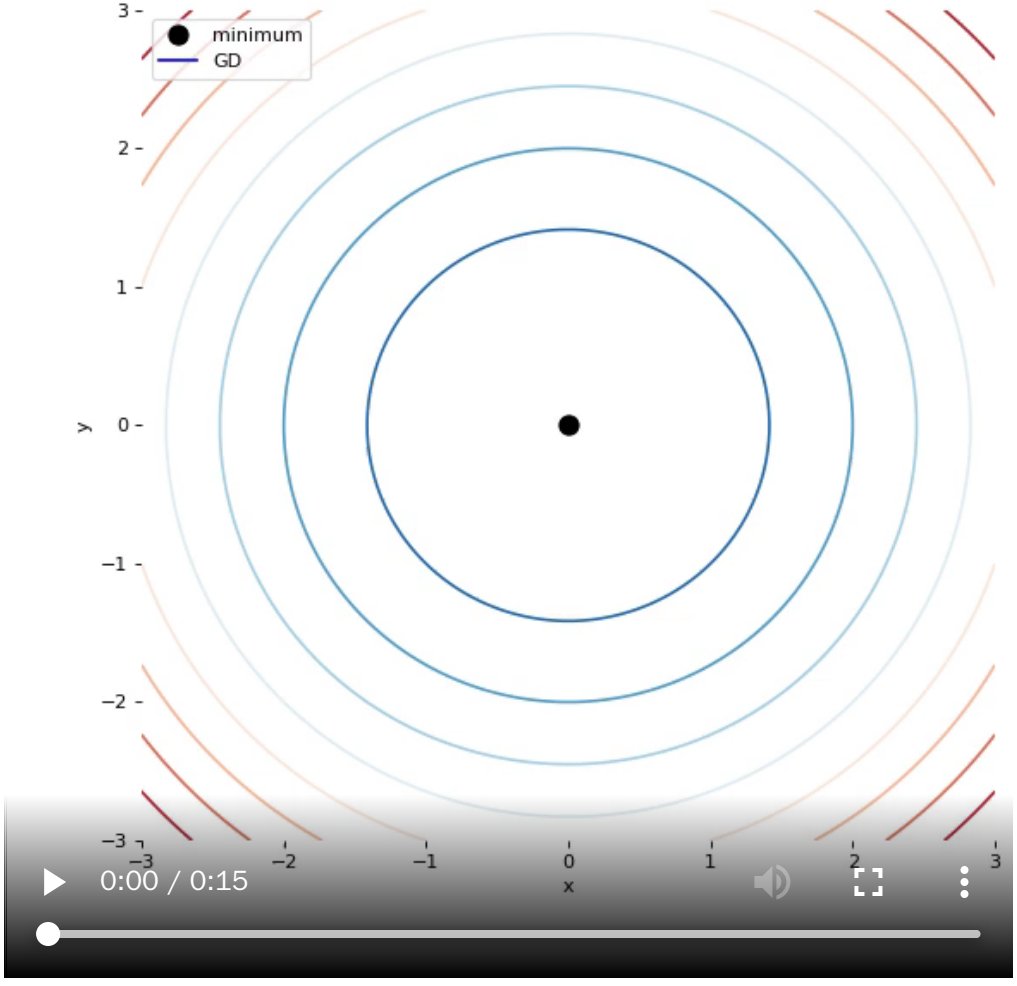
where the order $n(t, b)$ to visit the samples can be either sequential or random.

- Increasing the batch-size reduces the variance of the gradient estimates and enables the speed-up of batch processing.
- The stochastic behavior of this procedure helps evade local minima.

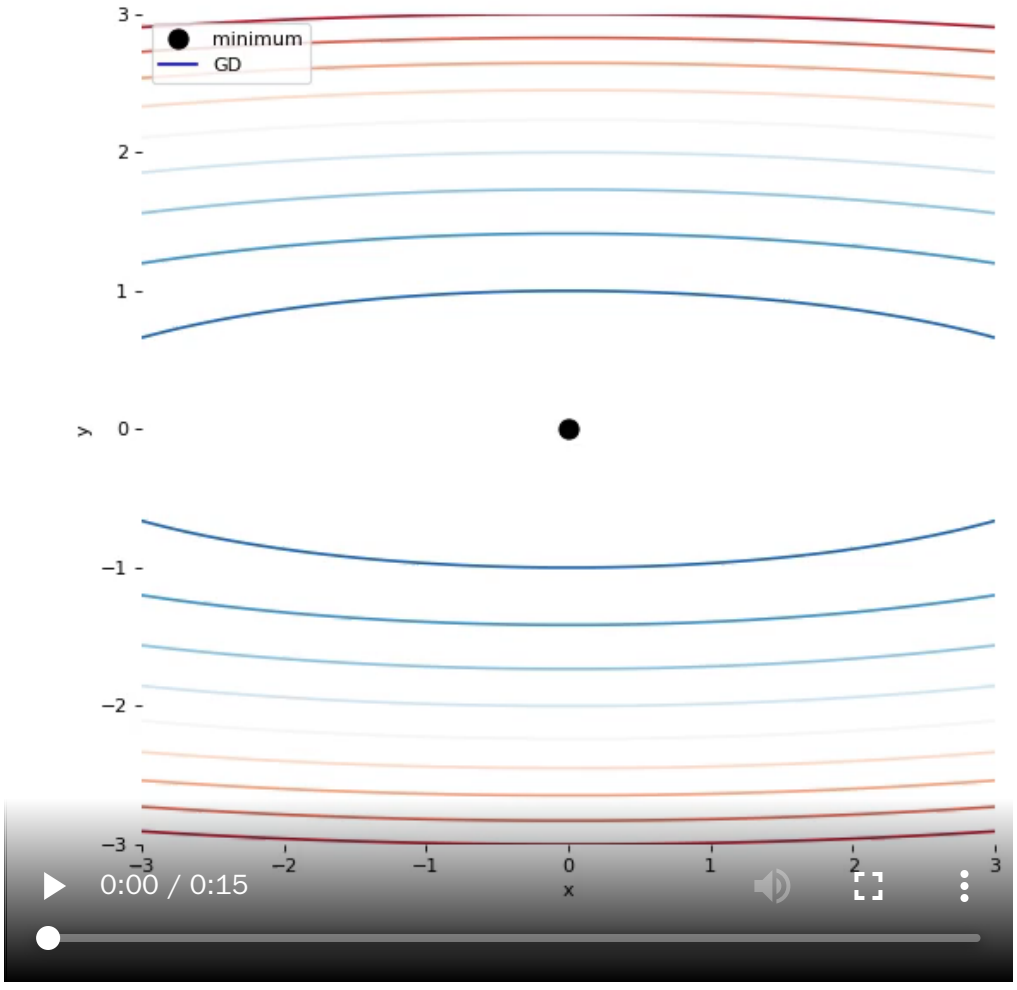
Limitations

The gradient descent method makes strong assumptions about

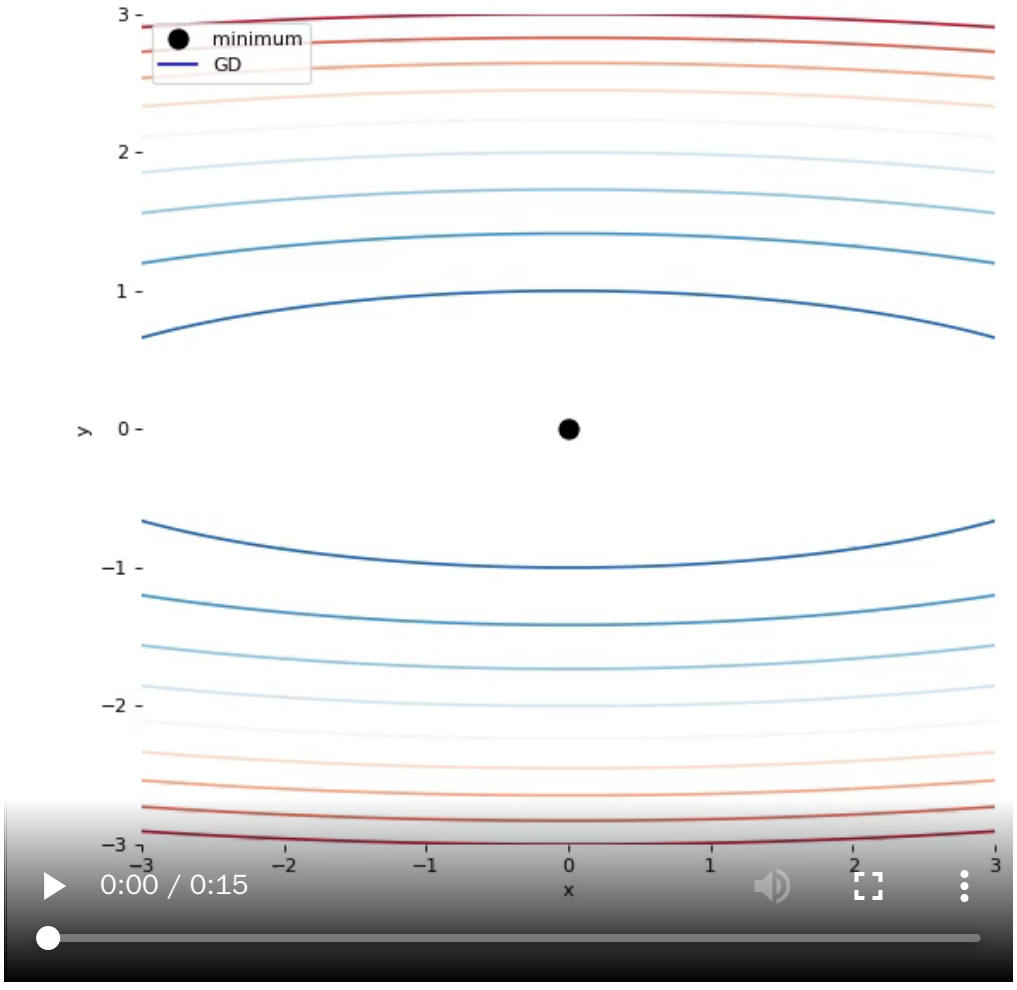
- the magnitude of the local curvature to set the step size,
- the isotropy of the curvature, so that the same step size γ makes sense in all directions.



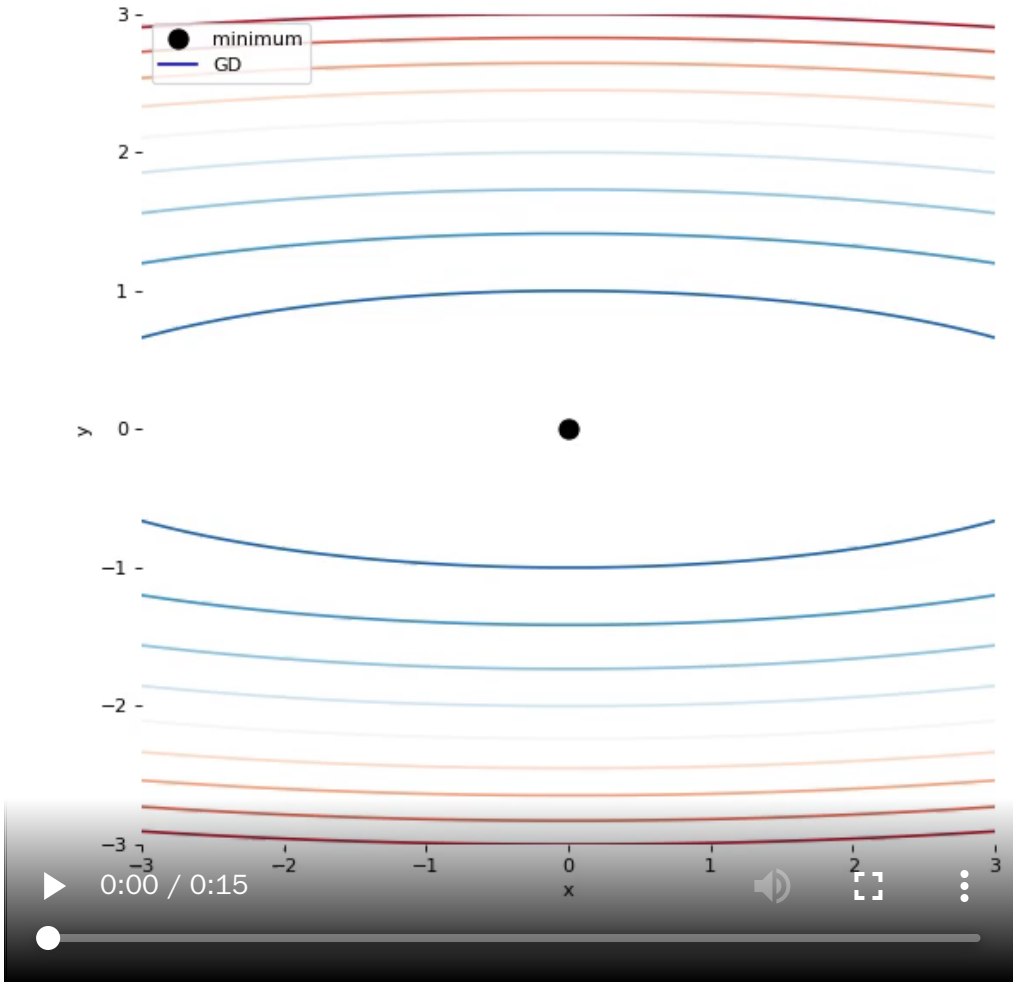
$\gamma = 0.01$



$\gamma = 0.01$



$\gamma = 0.1$



$\gamma = 0.4$

Wolfe conditions

Let us consider a function f to minimize along x, following a direction of descent p.

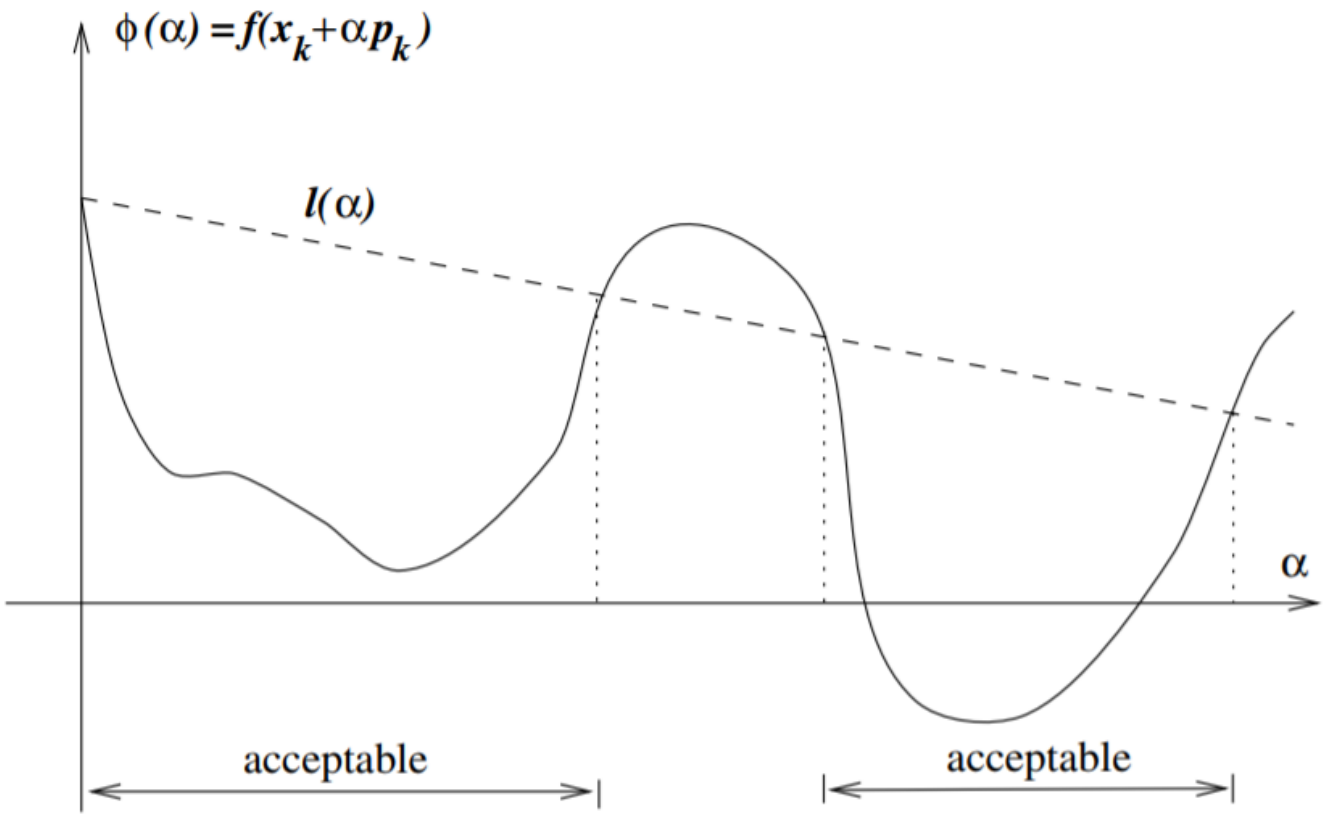
For $0 < c_1 < c_2 < 1$, the Wolfe conditions on the step size γ are as follows:

- Sufficient decrease condition:

$$f(x + \gamma p) \leq f(x) + c_1 \gamma p^T \nabla f(x)$$

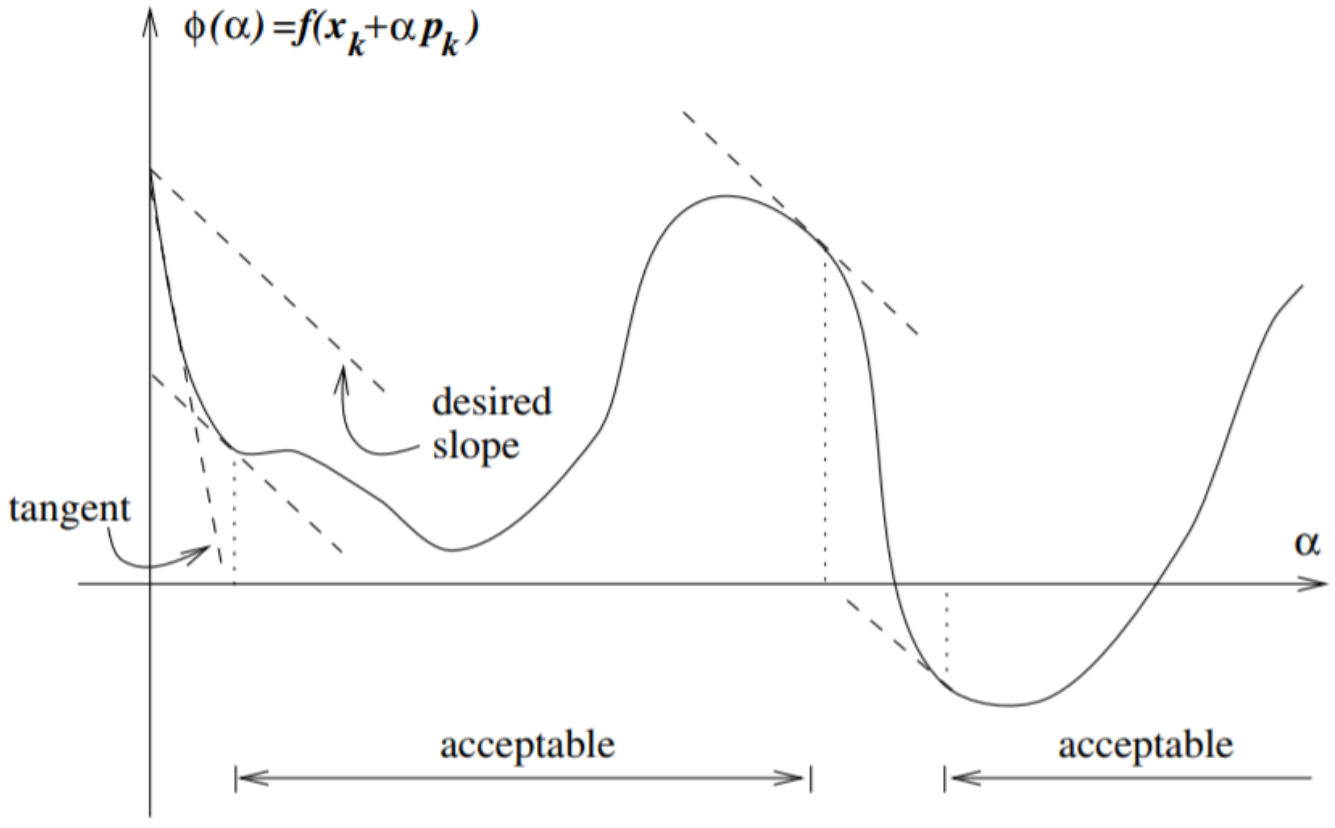
- Curvature condition:

$$c_2 p^T \nabla f(x) \leq p^T \nabla f(x + \gamma p)$$



The sufficient decrease condition ensures that f decreases sufficiently. (α is the step size.)

Credits: Wikipedia, Wolfe conditions.



The curvature condition ensures that the slope has been reduced sufficiently.

Credits: Wikipedia, Wolfe conditions.

The Wolfe conditions can be used to design line search algorithms to automatically determine a step size γ_t.

However, in deep learning,

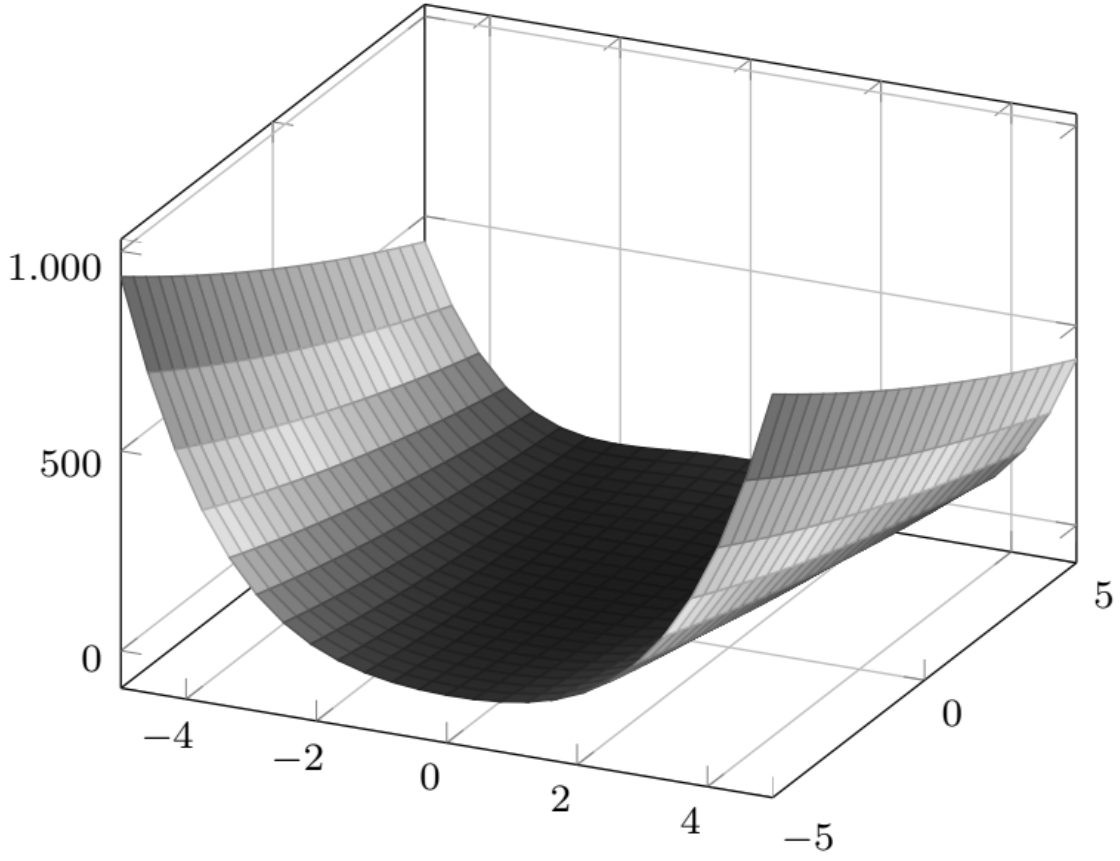
- these algorithms are impractical because of the size of the parameter space and the overhead it would induce,
- they might lead to overfitting when the empirical risk is minimized too well.

The tradeoffs of learning

When decomposing the excess error in terms of approximation, estimation and optimization errors, stochastic algorithms yield the best generalization performance (in terms of expected risk) despite being the worst optimization algorithms (in terms of empirical risk) (Bottou, 2011).

$$
\begin{aligned}
&\mathbb{E}\left[R(\tilde{f}_*^{\mathbf{d}}) - R(f_B)\right] \\
&= \mathbb{E}\left[R(f_*) - R(f_B)\right] + \mathbb{E}\left[R(f_*^{\mathbf{d}}) - R(f_*)\right] + \mathbb{E}\left[R(\tilde{f}_*^{\mathbf{d}}) - R(f_*^{\mathbf{d}})\right] \\
&= \mathcal{E}_{\text{app}} + \mathcal{E}_{\text{est}} + \mathcal{E}_{\text{opt}}
\end{aligned}
$$

Momentum

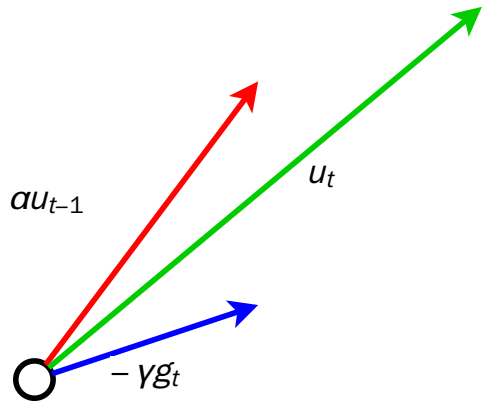


In the situation of small but consistent gradients, as through valley floors, gradient descent moves very slowly.

An improvement to gradient descent is to use momentum to add inertia in the choice of the step direction, that is

$$u_t = \alpha u_{t-1} - \gamma g_t$$
$$\theta_{t+1} = \theta_t + u_t.$$

- The new variable u_t is the velocity. It corresponds to the direction and speed by which the parameters move as the learning dynamics progresses, modeled as an exponentially decaying moving average of negative gradients.
- Gradient descent with momentum has three nice properties:
  - it can go through local barriers,
  - it accelerates if the gradient does not change much,
  - it dampens oscillations in narrow valleys.

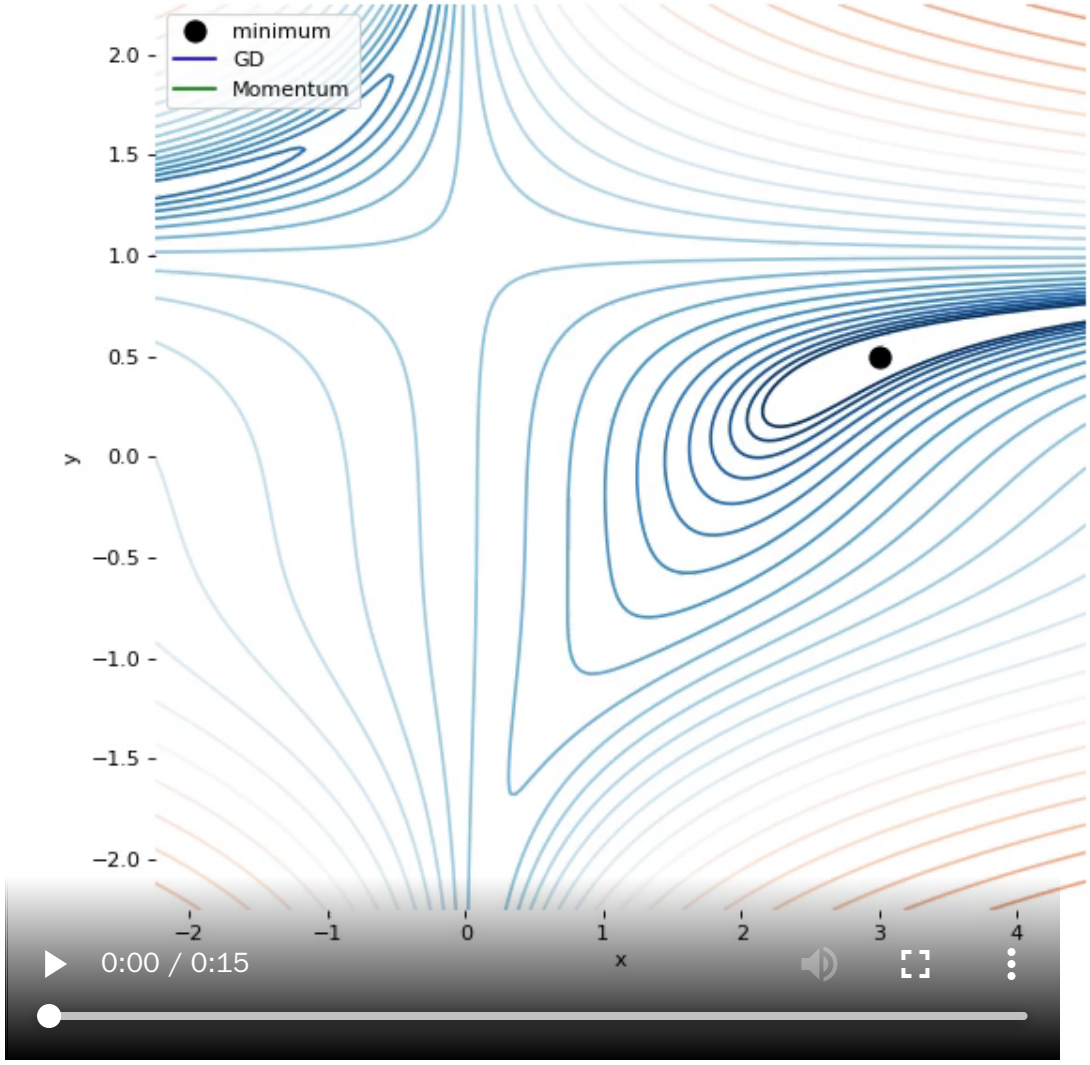


The hyper-parameter α controls how recent gradients affect the current update.

- Usually, $\alpha = 0.9$, with $\alpha > \gamma$.
- If at each update we observed g, the step would (eventually) be

$$u = -\frac{\gamma}{1-\alpha} g.$$

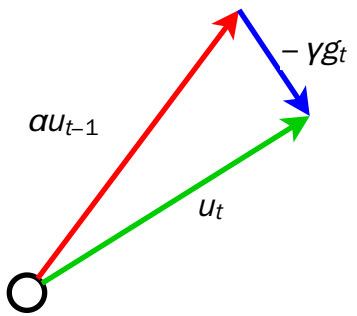
- Therefore, for $\alpha = 0.9$, it is like multiplying the maximum speed by 10 relative to the current direction.

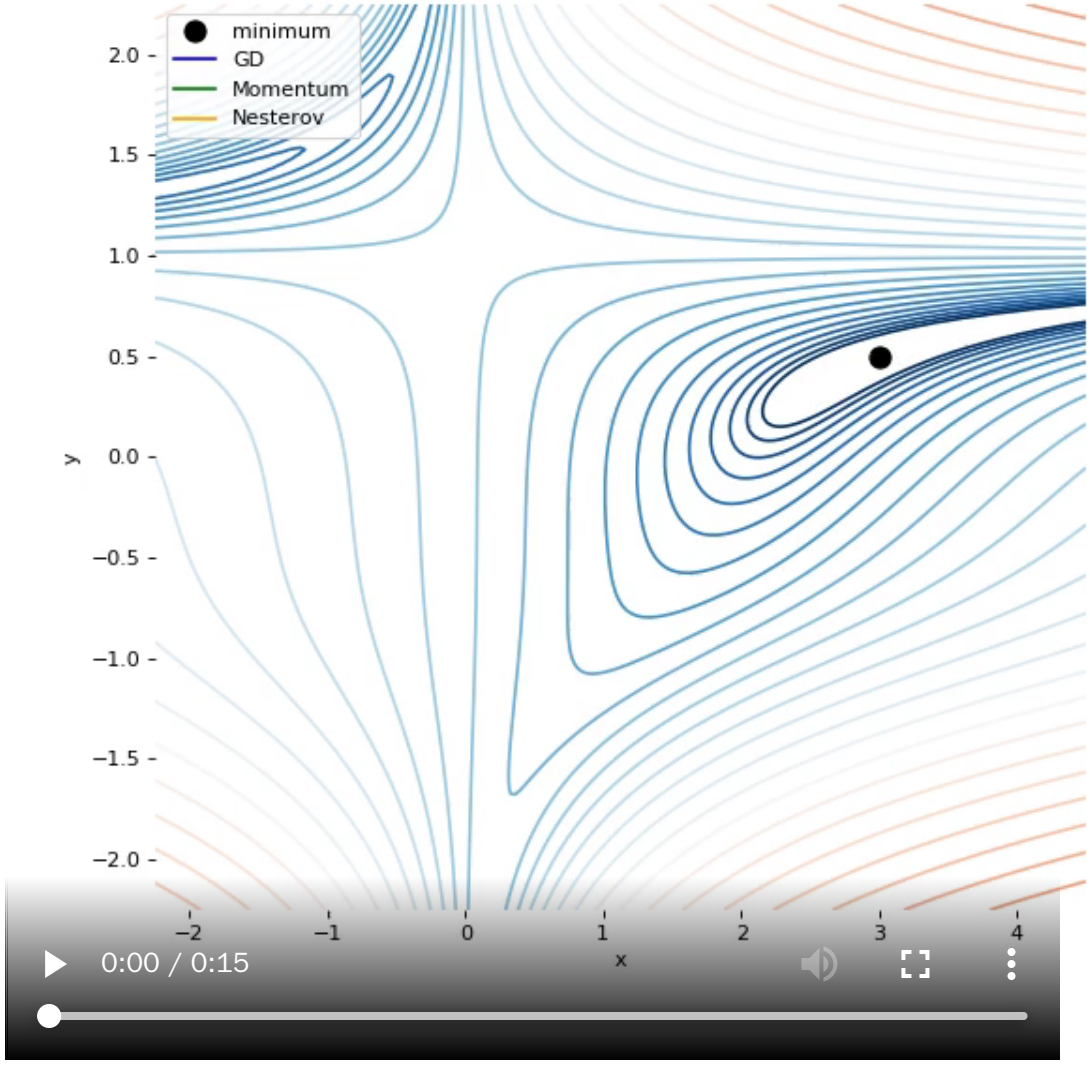

minimum
GD
Momentum
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2
−1
0
1
2
3
4
x
0:00 / 0:15

Nesterov momentum

An alternative consists in simulating a step in the direction of the velocity, then calculate the gradient and make a correction.

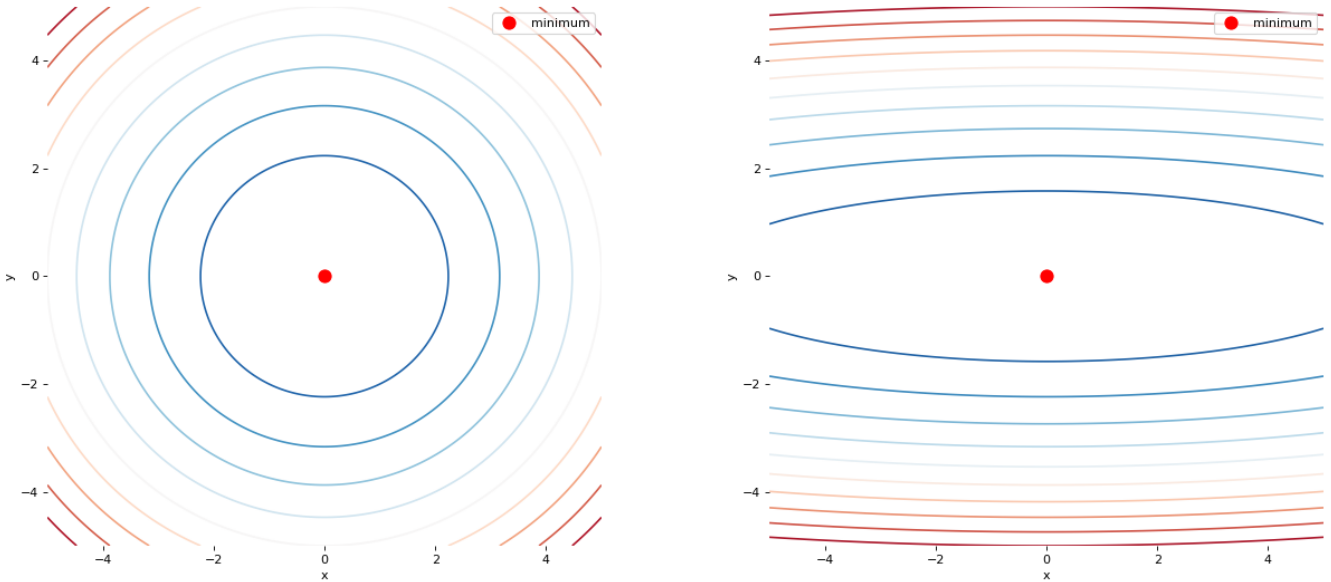
$$
\begin{aligned}
g_t &= \frac{1}{N} \sum_{n=1}^{N} \nabla_\theta \ell(y_n, f(\mathbf{x}_n; \theta_t + \alpha u_{t-1})) \\
u_t &= \alpha u_{t-1} - \gamma g_t \\
\theta_{t+1} &= \theta_t + u_t
\end{aligned}
$$




minimum
GD
Momentum
Nesterov
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2
−1
0
1
2
3
4
x
0:00 / 0:15

Adaptive learning rate

Vanilla gradient descent assumes the isotropy of the curvature, so that the same step size γ applies to all parameters.



Isotropic vs. Anistropic

AdaGrad

Per-parameter downscale by square-root of sum of squares of all its historical values.

$$r_t = r_{t-1} + g_t \odot g_t$$
$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{r_t}} \odot g_t.$$

- AdaGrad eliminates the need to manually tune the learning rate. Most implementation use $\gamma = 0.01$ as default.
- It is good when the objective is convex.
- r_t grows unboundedly during training, which may cause the step size to shrink and eventually become infinitesimally small.

RMSProp

Same as AdaGrad but accumulate an exponentially decaying average of the gradient.

$$r_t = \rho r_{t-1} + (1-\rho) g_t \odot g_t$$
$$\theta_{t+1} = \theta_t - \frac{\gamma}{\delta + \sqrt{r_t}} \odot g_t.$$

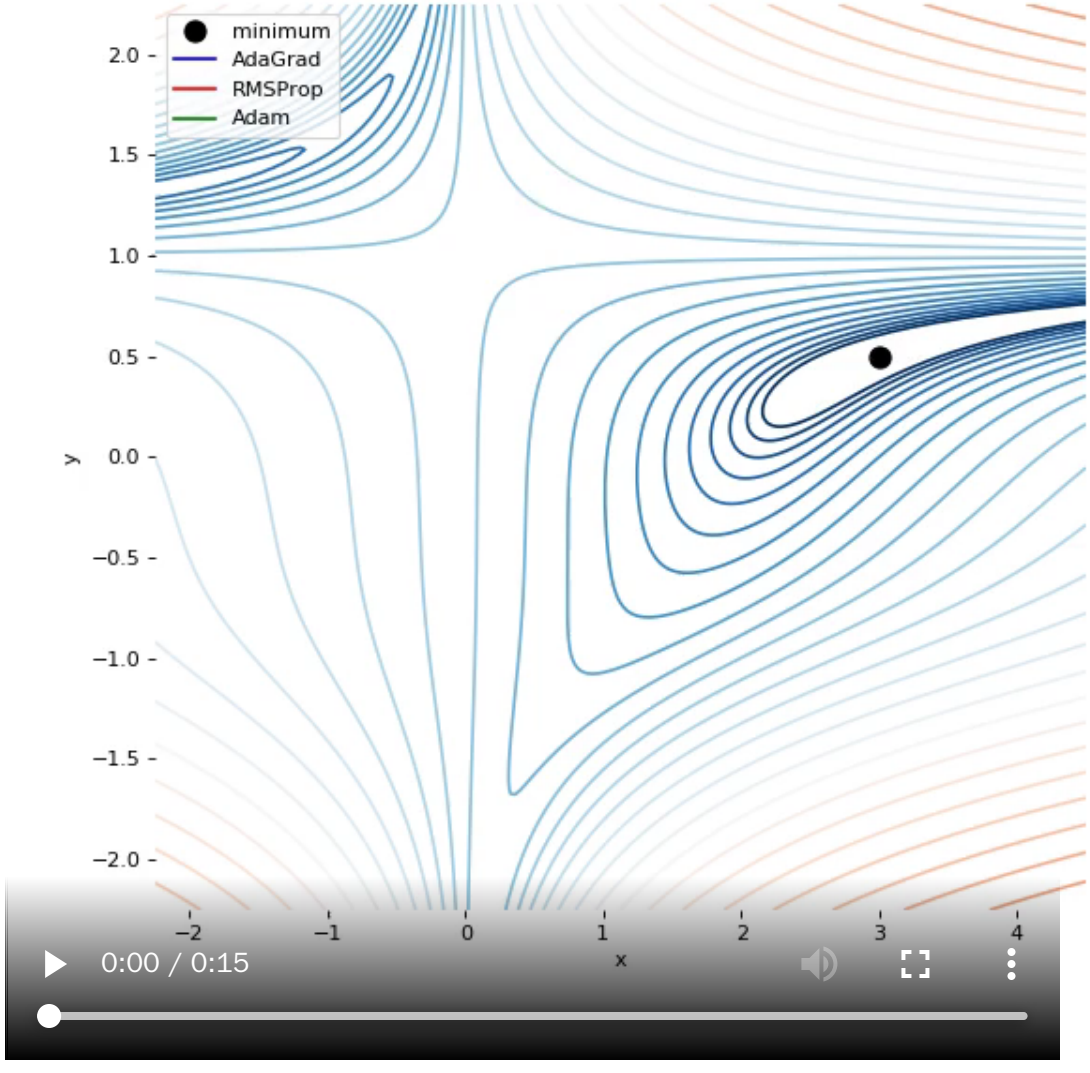
- Perform better in non-convex settings.
- Does not grow unboundedly.

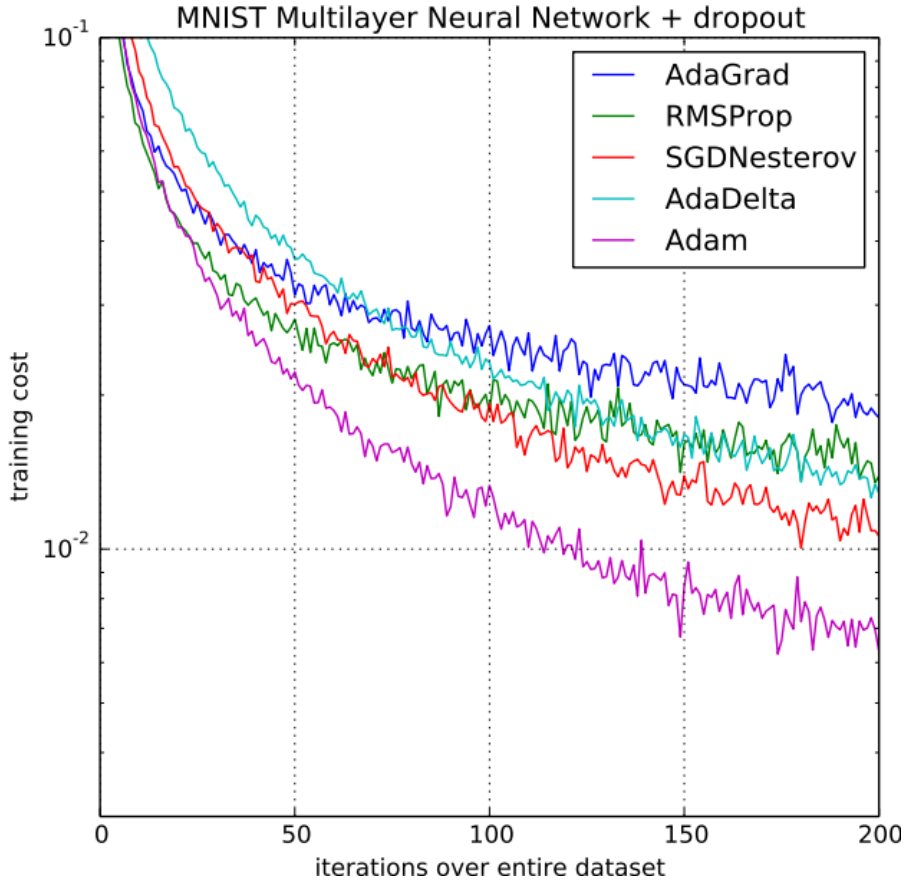
Adam

Similar to RMSProp with momentum, but with bias correction terms for the first and second moments.

$$
\begin{aligned}
\boldsymbol{s}_t &= \rho_1 \boldsymbol{s}_{t-1} + (1 - \rho_1) \boldsymbol{g}_t \\
\hat{\boldsymbol{s}}_t &= \frac{\boldsymbol{s}_t}{1 - \rho_1^t} \\
\boldsymbol{r}_t &= \rho_2 \boldsymbol{r}_{t-1} + (1 - \rho_2) \boldsymbol{g}_t \odot \boldsymbol{g}_t \\
\hat{\boldsymbol{r}}_t &= \frac{\boldsymbol{s}_t}{1 - \rho_2^t} \\
\boldsymbol{\theta}_{t+1} &= \boldsymbol{\theta}_t - \gamma \frac{\hat{\boldsymbol{s}}_t}{\delta + \sqrt{\hat{\boldsymbol{r}}_t}}
\end{aligned}
$$

- Good defaults are $\rho_1 = 0.9$ and $\rho_2 = 0.999$.
- Adam is one of the default optimizers in deep learning, along with SGD with momentum.


minimum
AdaGrad
RMSProp
Adam
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
y
−2
−1
0
1
2
3
4
x
0:00 / 0:15

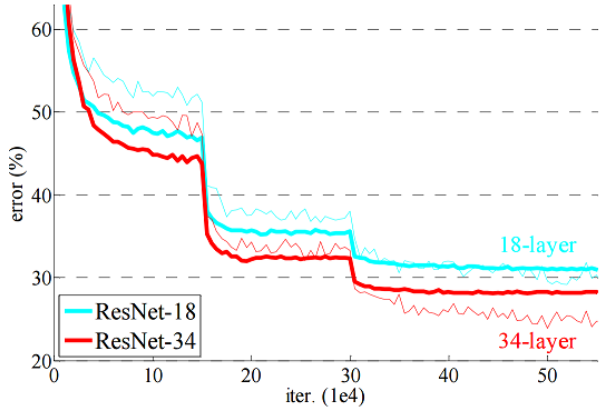


Credits: Kingma and Ba, Adam: A Method for Stochastic Optimization, 2014.

Scheduling

Despite per-parameter adaptive learning rate methods, it is usually helpful to anneal the learning rate γ over time.

- Step decay: reduce the learning rate by some factor every few epochs (e.g, by half every 10 epochs).
- Exponential decay: $\gamma_t = \gamma_0 \exp(-kt)$ where γ_0 and k are hyper-parameters.
- $1/t$ decay: $\gamma_t = \gamma_0/(1+kt)$ where γ_0 and k are hyper-parameters.



Step decay scheduling for training ResNets.

Initialization

- In convex problems, provided a good learning rate γ, convergence is guaranteed regardless of the initial parameter values.
- In the non-convex regime, initialization is much more important!
- Little is known on the mathematics of initialization strategies.
  - What is known: initialization should break symmetry.
  - What is known: the scale of weights is important.

Controlling for the variance in the forward pass

A first strategy is to initialize the network parameters such that activations preserve the same variance across layers.

Intuitively, this ensures that the information keeps flowing during the forward pass, without reducing or magnifying the magnitude of input signals exponentially.

Let us assume that

- we are in a linear regime at initialization (e.g., the positive part of a ReLU or the middle of a sigmoid),
- weights w_{ij}^l are initialized independently,
- biases b_l are initialized to be 0,
- input feature variances are the same, which we denote as $\mathbb{V}[x]$.

Then, the variance of the activation h_i^l of unit i in layer l is

$$\begin{aligned}\mathbb{V}\left[h_i^l\right] &= \mathbb{V}\left[\sum_{j=0}^{q_{l-1}-1} w_{ij}^l h_j^{l-1}\right] \\ &= \sum_{j=0}^{q_{l-1}-1} \mathbb{V}\left[w_{ij}^l\right] \mathbb{V}\left[h_j^{l-1}\right]\end{aligned}$$

where q_l is the width of layer l and $h_j^0 = x_j$ for all $j = 0, \ldots, p-1$.

If we further assume that weights w_{ij}^l at layer l share the same variance $\mathbb{V}\left[w^l\right]$ and that the variance of the activations in the previous layer are the same, then we can drop the indices and write

$$\mathbb{V}\left[h^l\right] = q_{l-1}\mathbb{V}\left[w^l\right]\mathbb{V}\left[h^{l-1}\right].$$

Therefore, the variance of the activations is preserved across layers when

$$\mathbb{V}\left[w^l\right] = \frac{1}{q_{l-1}} \quad \forall l.$$

This condition is enforced in LeCun's uniform initialization, which is defined as

$$w_{ij}^l \sim \mathcal{U}\left[-\sqrt{\frac{3}{q_{l-1}}}, \sqrt{\frac{3}{q_{l-1}}}\right].$$

Controlling for the variance in the backward pass

A similar idea can be applied to ensure that the gradients flow in the backward pass (without vanishing nor exploding), by maintaining the variance of the gradient with respect to the activations fixed across layers.

Under the same assumptions as before,

$$\begin{aligned}
\mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h_i^l}\right] &= \mathbb{V}\left[\sum_{j=0}^{q_{l+1}-1} \frac{\mathrm{d}\hat{y}}{\mathrm{d}h_j^{l+1}} \frac{\partial h_j^{l+1}}{\partial h_i^l}\right] \\
&= \mathbb{V}\left[\sum_{j=0}^{q_{l+1}-1} \frac{\mathrm{d}\hat{y}}{\mathrm{d}h_j^{l+1}} w_{j,i}^{l+1}\right] \\
&= \sum_{j=0}^{q_{l+1}-1} \mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h_j^{l+1}}\right] \mathbb{V}\left[w_{ji}^{l+1}\right]
\end{aligned}$$

If we further assume that

- the gradients of the activations at layer l share the same variance
- the weights at layer $l+1$ share the same variance $\mathbb{V}\left[w^{l+1}\right]$,

then we can drop the indices and write

$$\mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h^{l}}\right] = q_{l+1}\mathbb{V}\left[\frac{\mathrm{d}\hat{y}}{\mathrm{d}h^{l+1}}\right]\mathbb{V}\left[w^{l+1}\right].$$

Therefore, the variance of the gradients with respect to the activations is preserved across layers when

$$\mathbb{V}\left[w^{l}\right] = \frac{1}{q_{l}} \quad \forall l.$$

Xavier initialization

We have derived two different conditions on the variance of w^l,

- $\mathbb{V}\left[w^l\right] = \frac{1}{q_{l-1}}$
- $\mathbb{V}\left[w^l\right] = \frac{1}{q_l}$.

A compromise is the Xavier initialization, which initializes w^l randomly from a distribution with variance

$$\mathbb{V}\left[w^l\right] = \frac{1}{\frac{q_{l-1}+q_l}{2}} = \frac{2}{q_{l-1}+q_l}.$$

For example, normalized initialization is defined as

$$w_{ij}^l \sim \mathcal{U}\left[-\sqrt{\frac{6}{q_{l-1}+q_l}}, \sqrt{\frac{6}{q_{l-1}+q_l}}\right].$$

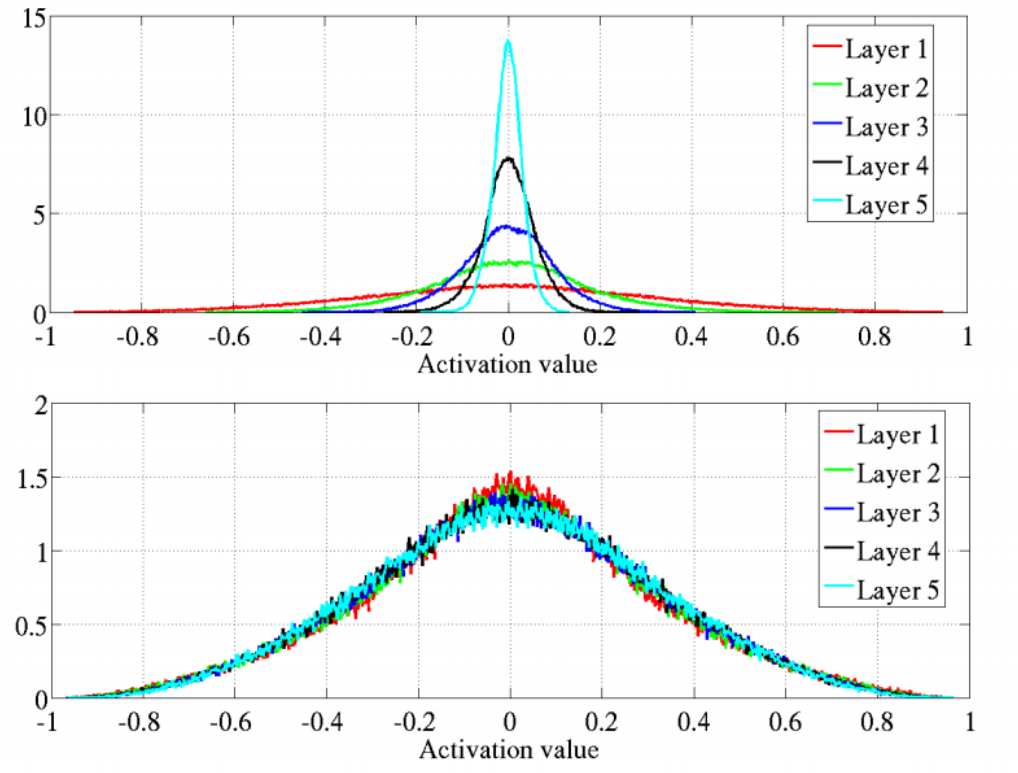


Figure 6: *Activation values normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized initialization (bottom). Top: 0-peak increases for higher layers.*

Credits: Glorot and Bengio, Understanding the difficulty of training deep feedforward neural networks, 2010.

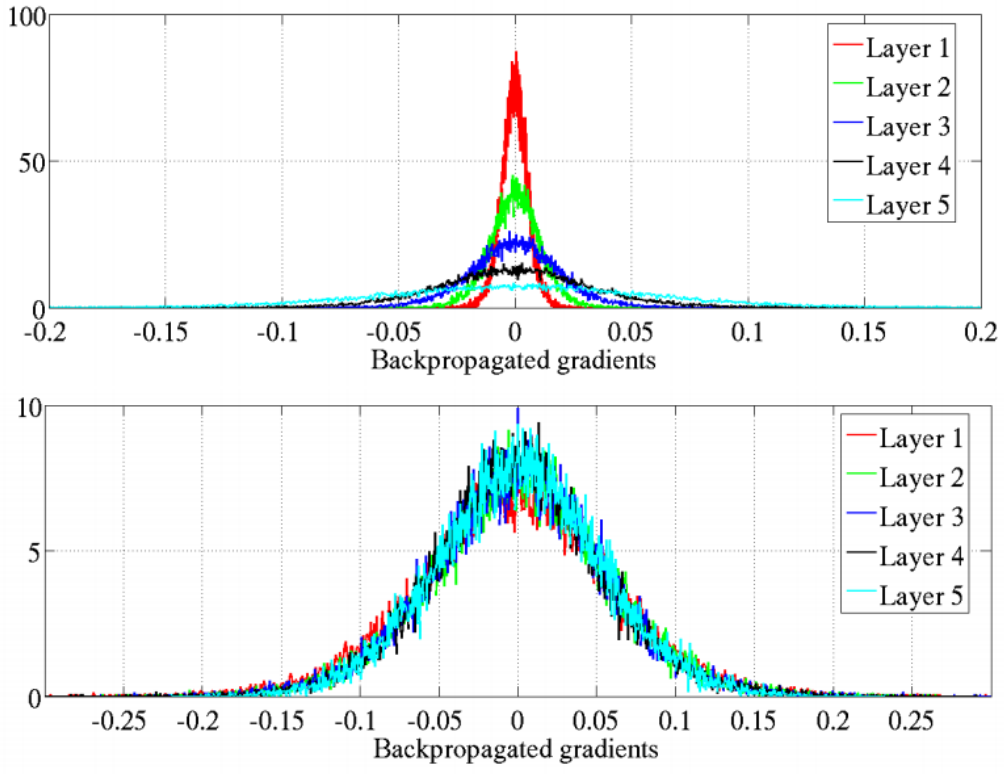


Figure 7: *Back-propagated gradients normalized histograms with hyperbolic tangent activation, with standard (top) vs normalized (bottom) initialization. Top: 0-peak decreases for higher layers.*

Credits: Glorot and Bengio, Understanding the difficulty of training deep feedforward neural networks, 2010.

Normalization

Data normalization

Previous weight initialization strategies rely on preserving the activation variance constant across layers, under the initial assumption that the input feature variances are the same.

That is,

$$\mathbb{V}\left[x_i\right] = \mathbb{V}\left[x_j\right] \triangleq \mathbb{V}\left[x\right]$$

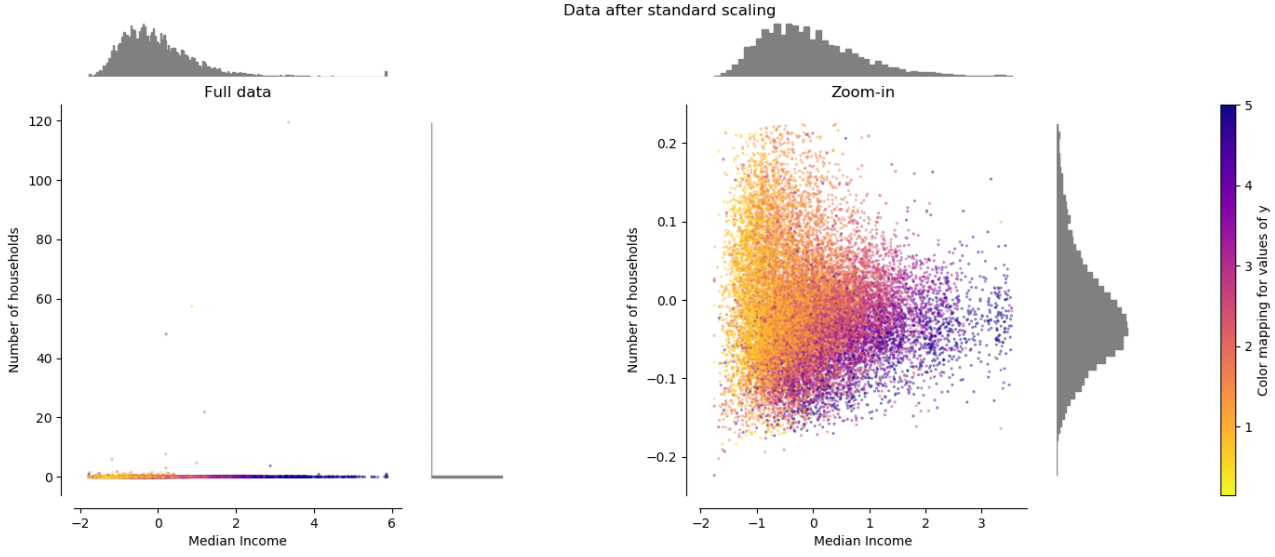
for all pairs of features i, j.

In general, this constraint is not satisfied but can be enforced by standardizing the input data feature-wise,

$$\mathbf{x}' = (\mathbf{x} - \hat{\mu}) \odot \frac{1}{\hat{\sigma}},$$

where

$$\hat{\mu} = \frac{1}{N} \sum_{\mathbf{x} \in \mathbf{d}} \mathbf{x} \qquad \hat{\sigma}^2 = \frac{1}{N} \sum_{\mathbf{x} \in \mathbf{d}} (\mathbf{x} - \hat{\mu})^2.$$



Credits: Scikit-Learn, Compare the effect of different scalers on data with outliers.

Batch normalization

Maintaining proper statistics of the activations and derivatives is critical for training neural networks.

- This constraint can be enforced explicitly during the forward pass by re-normalizing them.
- Batch normalization was the first method introducing this idea.

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

Abstract

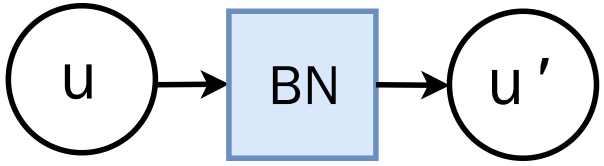
Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This slows down the training by requiring lower learning rates and careful parameter initialization, and makes it notoriously hard to train models with saturating nonlinearities. We refer to this phenomenon as *internal covariate*

Using mini-batches of examples, as opposed to one example at a time, is helpful in several ways. First, the gradient of the loss over a mini-batch is an estimate of the gradient over the training set, whose quality improves as the batch size increases. Second, computation over a batch can be much more efficient than m computations for individual examples, due to the parallelism afforded by the modern computing platforms.

While stochastic gradient is simple and effective, it

Credits: Ioffe and Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

- During training, batch normalization shifts and rescales according to the mean and variance estimated on the batch.
- During test, it shifts and rescales according to the empirical moments estimated during training.



Let us consider a given minibatch of samples at training, for which $\mathbf{u}_b \in \mathbb{R}^q$, $b = 1, \ldots, B$, are intermediate values computed at some location in the computational graph.

In batch normalization following the node $\mathbf{u}$, the per-component mean and variance are first computed on the batch

$$\hat{\mu}_{\text{batch}} = \frac{1}{B}\sum_{b=1}^{B} \mathbf{u}_b \qquad \hat{\sigma}^2_{\text{batch}} = \frac{1}{B}\sum_{b=1}^{B} (\mathbf{u}_b - \hat{\mu}_{\text{batch}})^2,$$

from the which the standardized $\mathbf{u}'_b \in \mathbb{R}^q$ are computed such that

$$\mathbf{u}'_b = \gamma \odot (\mathbf{u}_b - \hat{\mu}_{\text{batch}}) \odot \frac{1}{\hat{\sigma}_{\text{batch}} + \epsilon} + \beta$$

where $\gamma, \beta \in \mathbb{R}^q$ are parameters to optimize.

Exercise: How does batch normalization combine with backpropagation?

During inference, batch normalization shifts and rescales each component according to the empirical moments estimated during training:

$$\mathbf{u}' = \gamma \odot (\mathbf{u} - \hat{\mu}) \odot \frac{1}{\hat{\sigma}} + \beta.$$

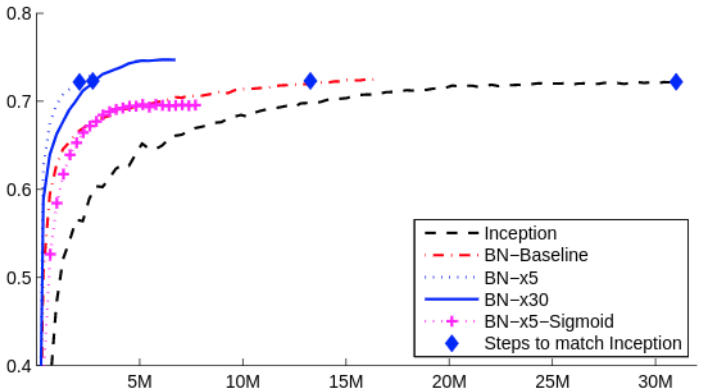


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

Credits: Ioffe and Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

The position of batch normalization relative to the non-linearity is not clear.

where W and b are learned parameters of the model, and $g(\cdot)$ is the nonlinearity such as sigmoid or ReLU. This formulation covers both fully-connected and convolutional layers. We add the BN transform immediately before the nonlinearity, by normalizing $\mathrm{x} = W\mathrm{u}+\mathrm{b}$. We could have also normalized the layer inputs u, but since u is likely the output of another nonlinearity, the shape of its distribution is likely to change during training, and constraining its first and second moments would not eliminate the covariate shift. In contrast, $W\mathrm{u} + \mathrm{b}$ is more likely to have a symmetric, non-sparse distribution, that is "more Gaussian" (Hyvärinen & Oja, 2000); normalizing it is likely to produce activations with a stable distribution.

Credits: Ioffe and Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015.

Layer normalization

Given a single input sample $\mathbf{x}$, a similar approach can be applied to standardize the activations $\mathbf{u}$ across a layer instead of doing it over the batch.

The end.