Deep Learning

Lecture 4: Around optimization and optimizers for neural nets

Prof. Stéphane Gaïffas
https://stephanegaiffas.github.io

Agenda for today

1. Batch versus stochastic gradient descent
2. Momentum methods
3. Adaptive learning rates
4. Learning rate scheduling

Batch versus stochastic gradient descent

Batch gradient descent

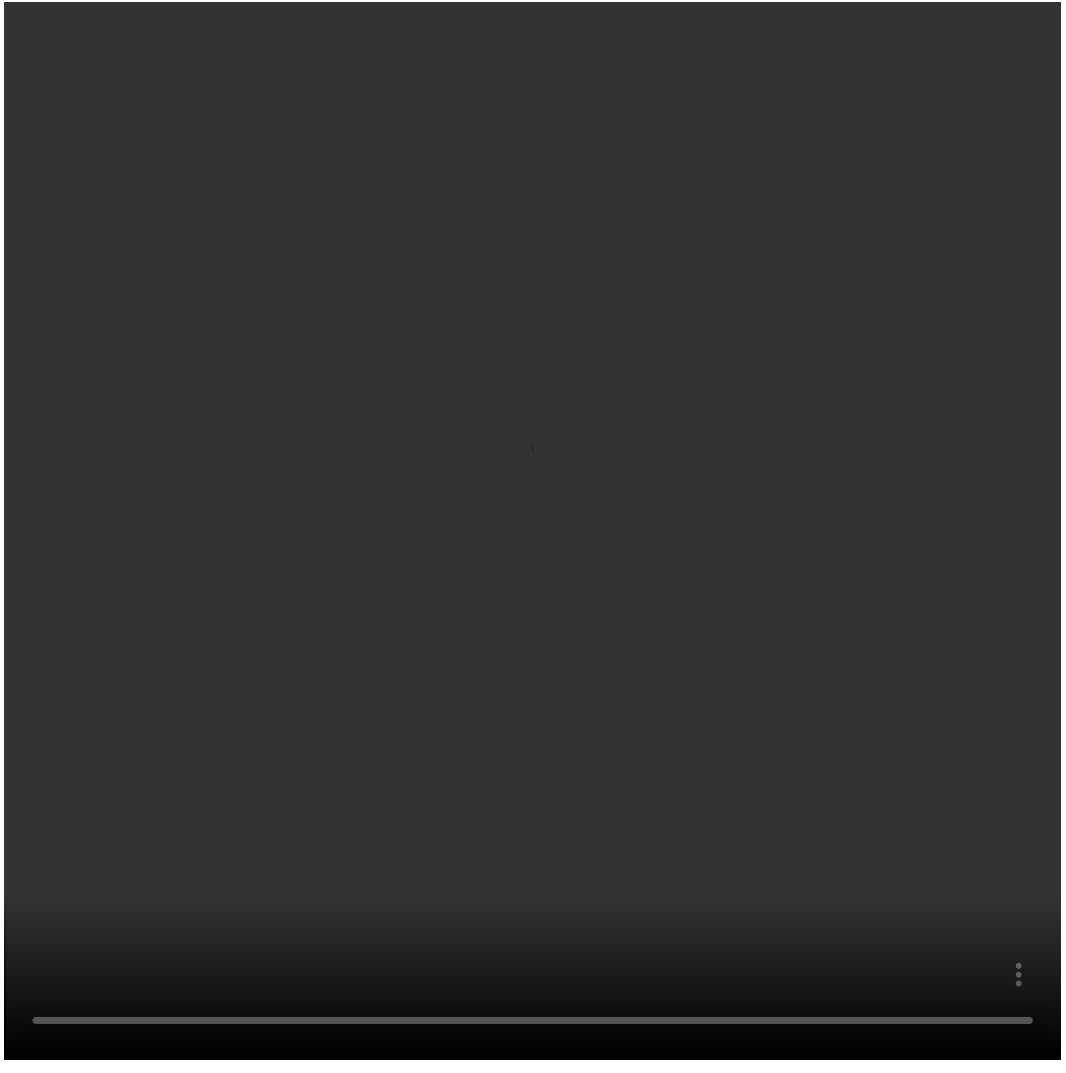
To minimize a goodness-of-fit function $R_n(\theta)$ of the form

$$R_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(x_i, \theta)),$$

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

$$\begin{aligned} g_t &= \frac{1}{n} \sum_{i=1}^{n} \nabla_\theta \ell(y_i, f(x_i; \theta_t)) \\ \theta_{t+1} &= \theta_t - \gamma g_t, \end{aligned}$$

where γ is the learning rate.



Batch gradient descent

While it makes sense 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): any partial sum is also an unbiased estimate, although of greater variance.

Consider an ideal case where the training dataset is the same set of $m \ll n$ samples replicated k times. Then,

$$R_n(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f(x_i, \theta)) = \frac{k}{n} \sum_{l=1}^{m} \ell(y_l, f(x_l; \theta)).$$

Instead of summing over n samples and moving by γ, we can use only m samples and move by $k\gamma$: this devides the computation by k.

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

Stochastic gradient descent

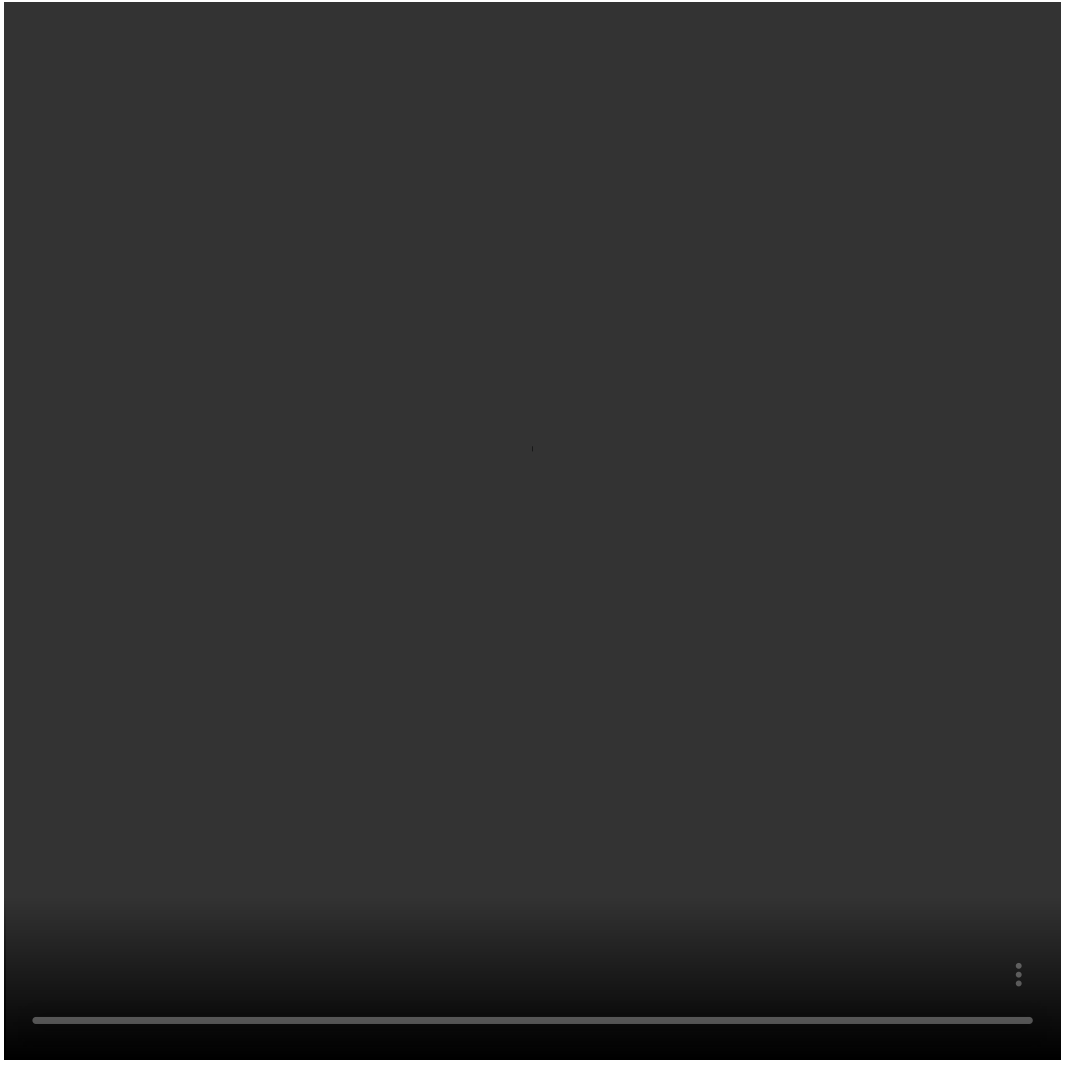
To reduce the computational complexity, stochastic gradient descent (SGD) consists in updating the parameters after every sample

$$\begin{aligned} g_t &= \nabla_\theta \ell(y_{i(t)}, f(x_{i(t)}; \theta_t)) \\ \theta_{t+1} &= \theta_t - \gamma g_t \end{aligned}$$

where $i(t)$ is sampled uniformly in $\{1, \dots, n\}$ at each iteration. The stochastic behavior of SGD helps to evade local minima.

While being computationally faster than batch gradient descent,

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



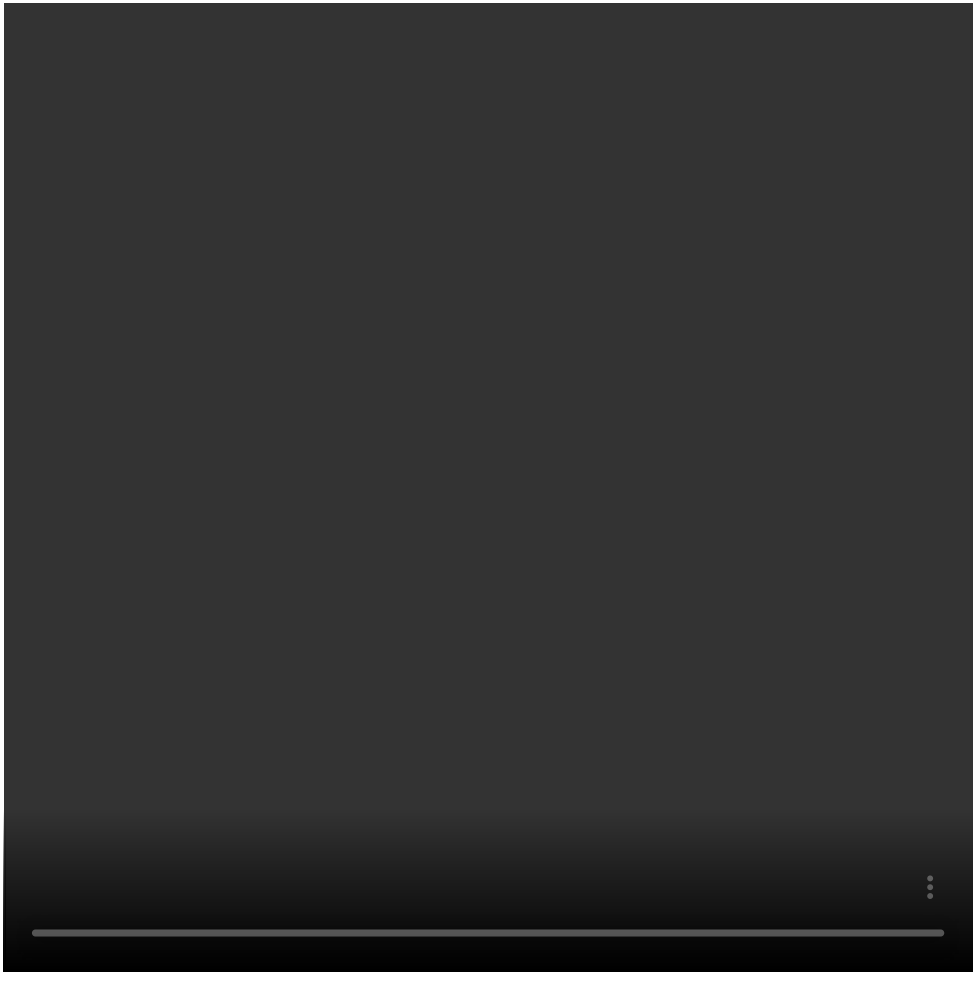
Instead, mini-batch SGD uses samples in mini-batches to update the parameters at each step

$$\theta_{t+1} = \theta_t - \gamma g_t \quad \text{where} \quad g_t = \frac{1}{|B_t|} \sum_{i \in B_t} \nabla_\theta \ell(y_i, f(x_i; \theta_t))$$

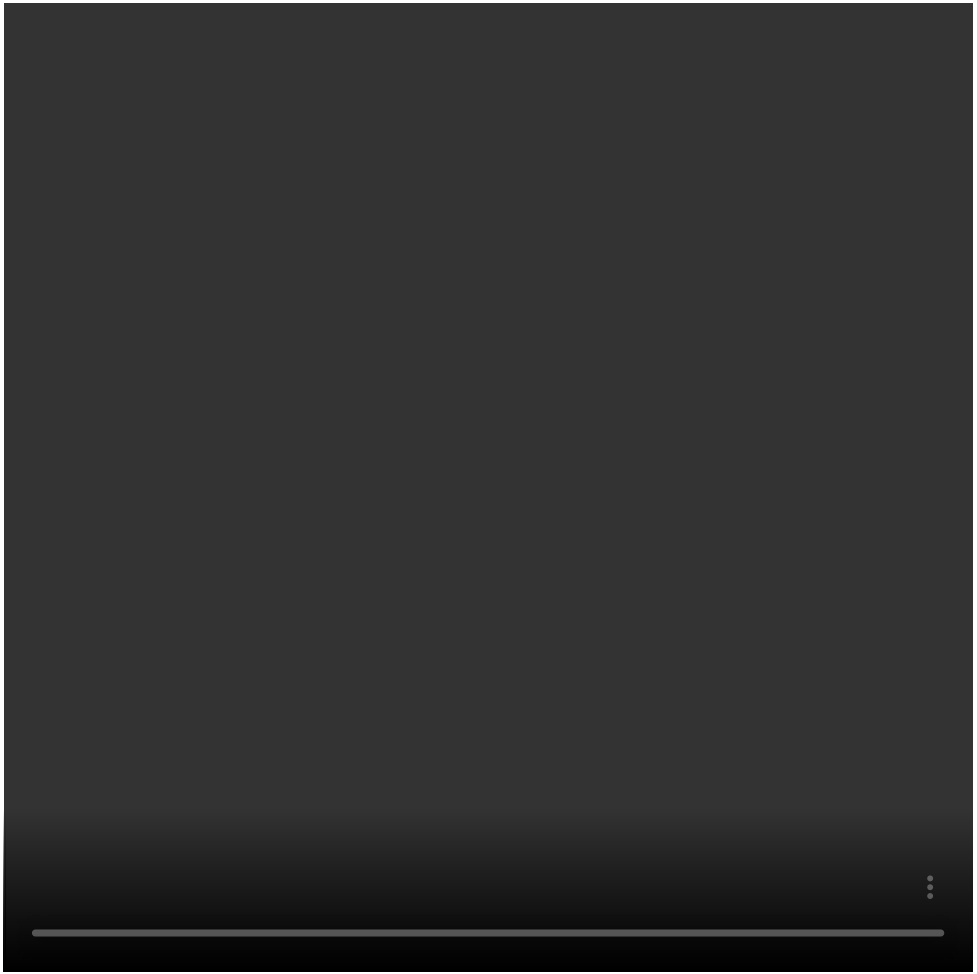
where each mini-batch B_t is typically obtained by reading sequentially shuffled data, meaning that a random permutation is applied on the data at the beginning of each epoch

Remarks

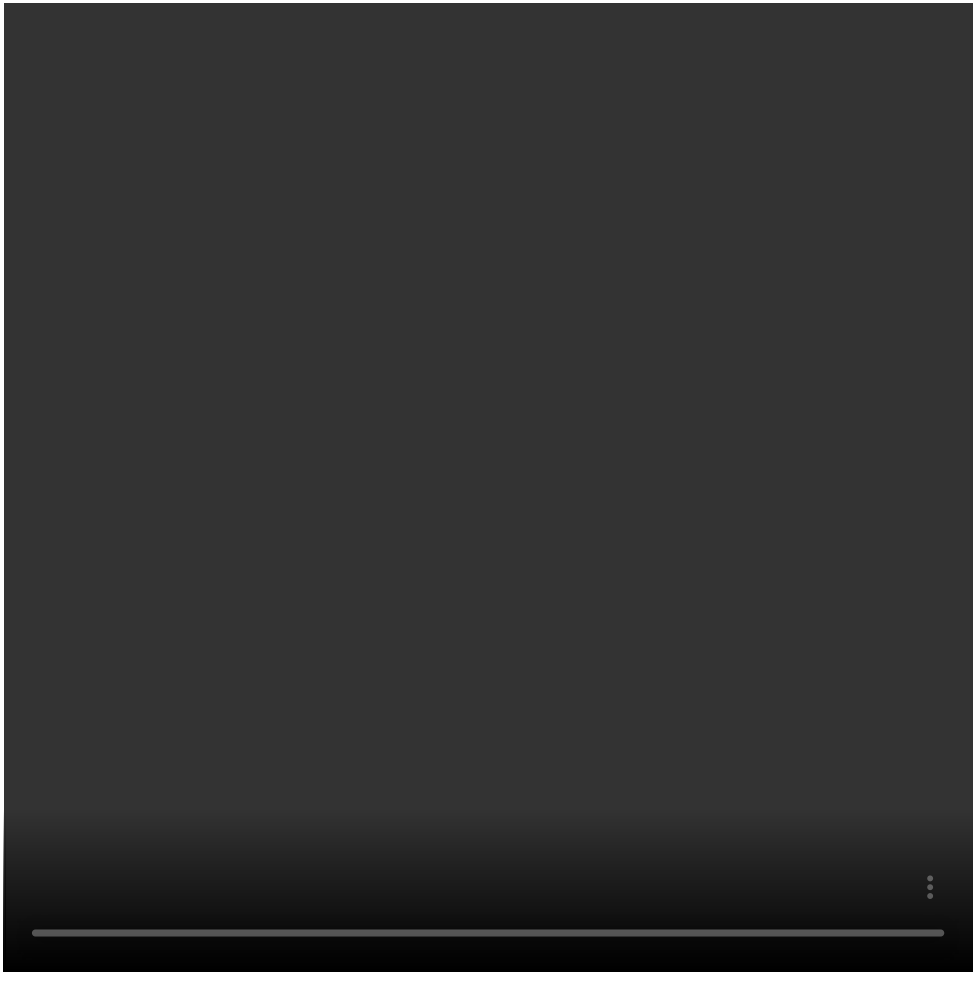
- Increasing the batch size B reduces the variance of the gradient estimates and gives a speed-up (more parallel computations)
- The interplay between B and γ is unclear
- Gradient descent makes strong assumptions about the magnitude of the local curvature to set the step size and the isotropy of the curvature, so that the same step size γ makes sense in all directions.



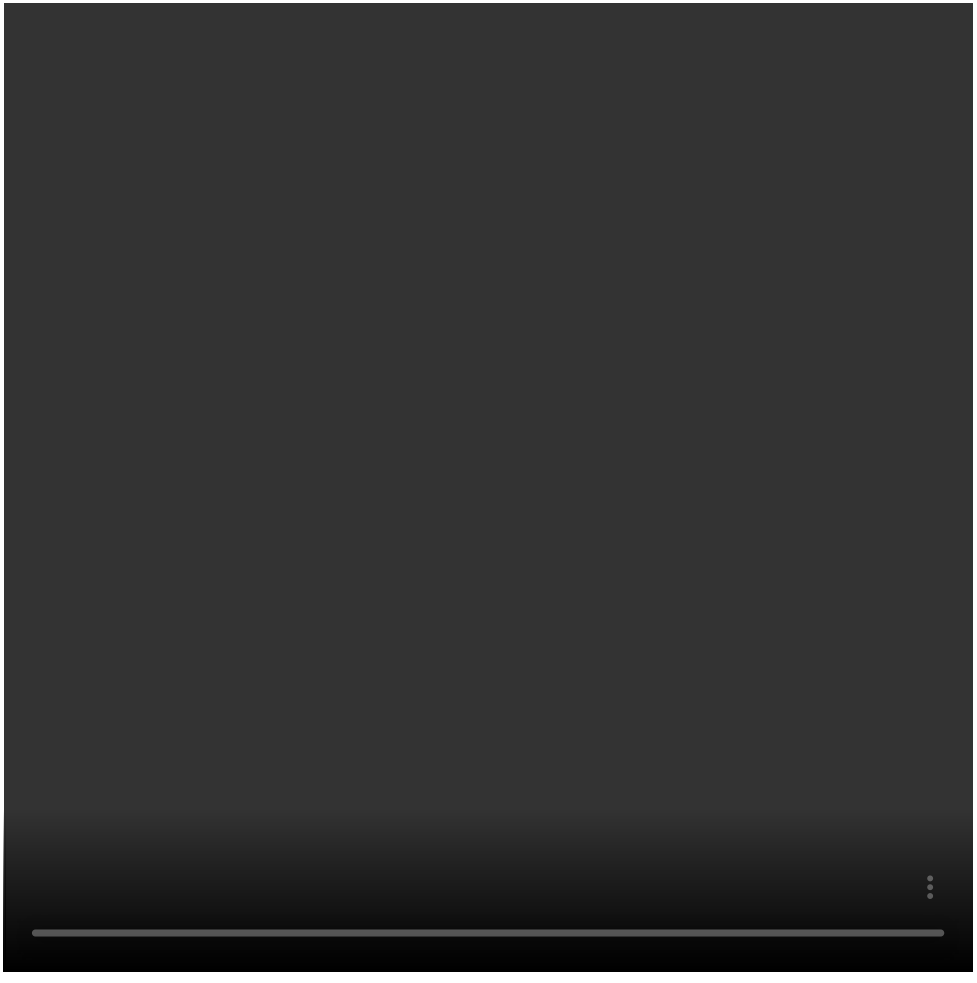
$\gamma = 0.01$



$\gamma = 0.01$



$\gamma = 0.1$

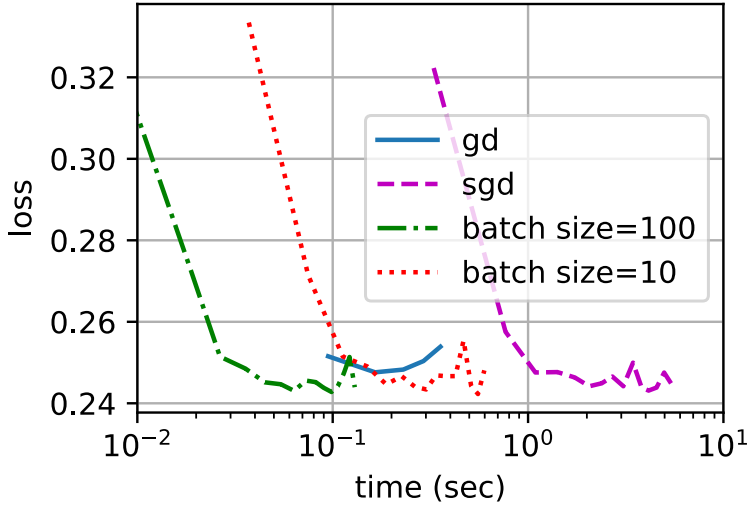


$\gamma = 0.4$

Tradeoffs of large-scale learning

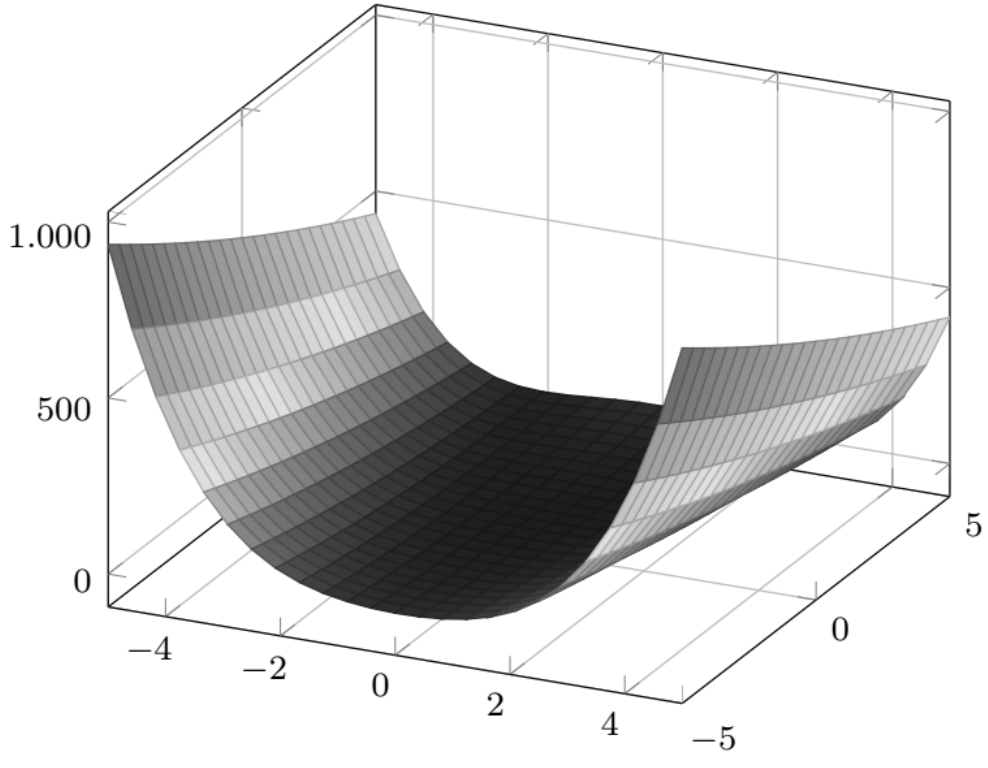
Bottou and Bousquet (2011): stochastic optimization algorithms (e.g., SGD) yield the best generalization performance (in terms of excess error) despite being the worst optimization algorithms for minimizing the empirical risk.

For a fixed computational budget, stochastic optimization algorithms reach a lower test error than more sophisticated algorithms (2nd order methods, line search algorithms, etc) that would fit the training error too well



Momentum methods

Momentum

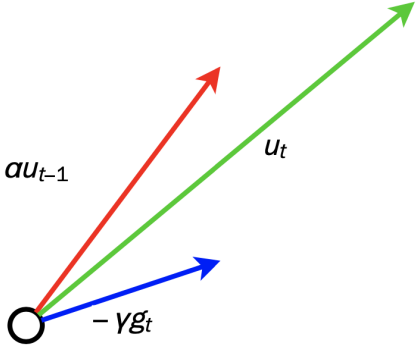


In the situation of small but consistent gradients, as through valley floors, gradient descent moves very slowly.

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

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

New variable u_t is the velocity. It's the direction and speed by which parameters move during training, 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
- https://distill.pub/2017/momentum/

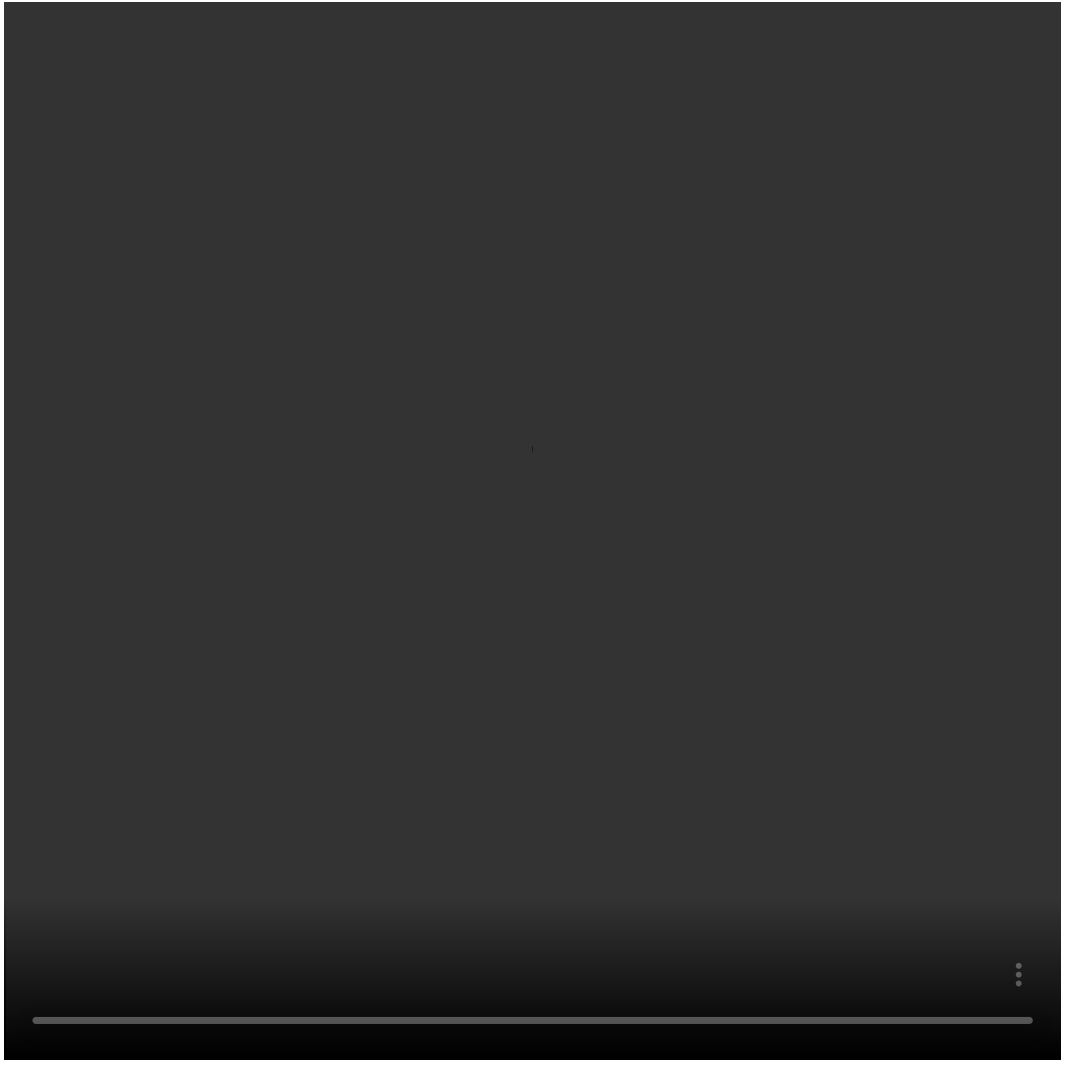
"Some methods of speeding up the convergence of iteration methods", Polyak 1964

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.

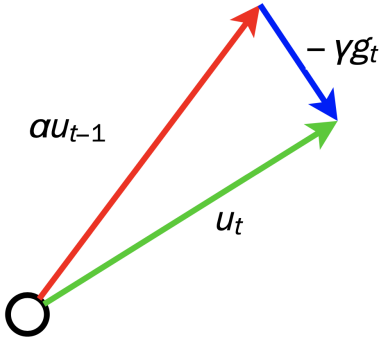


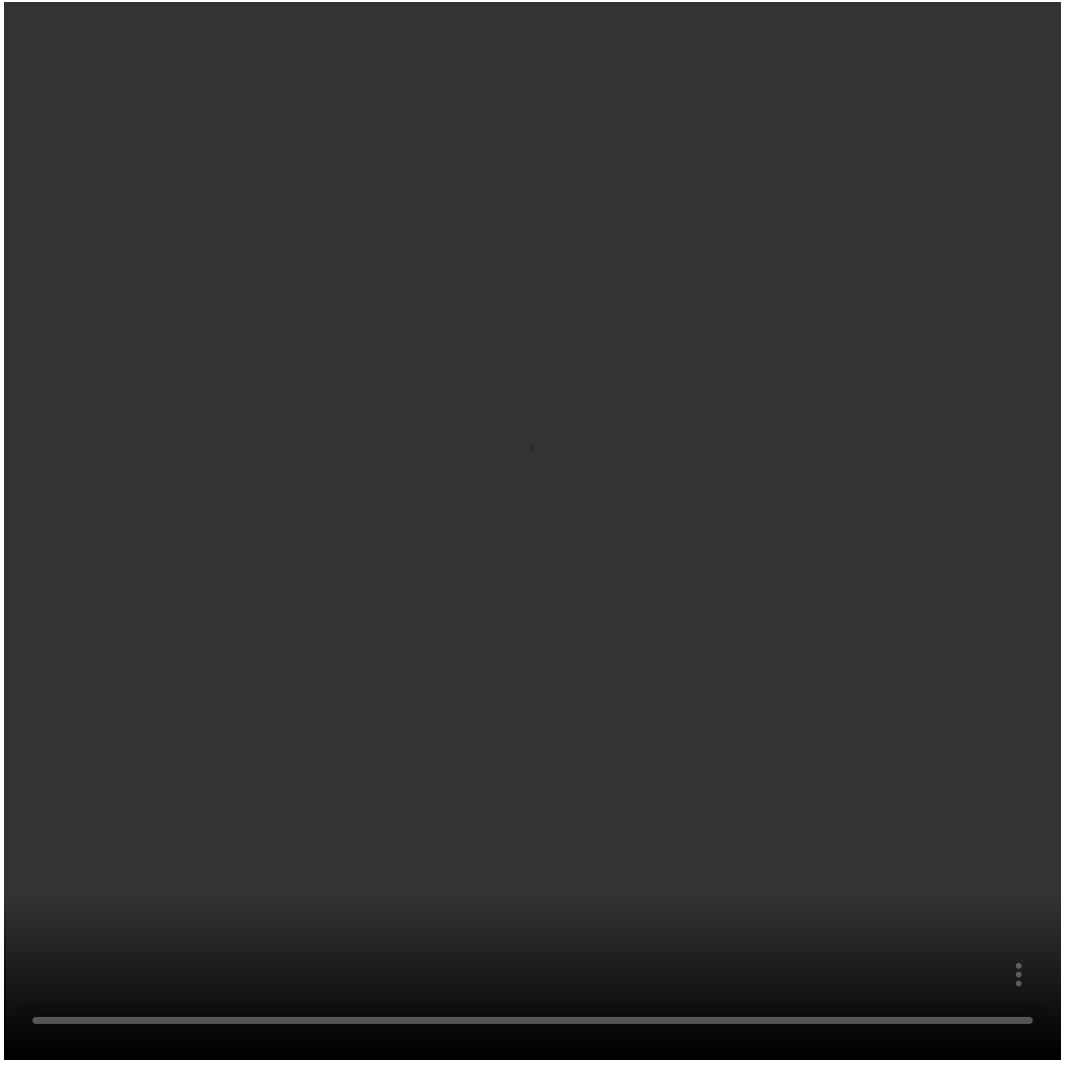
Nesterov momentum

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

It's an improvement over Polyak's momentum

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

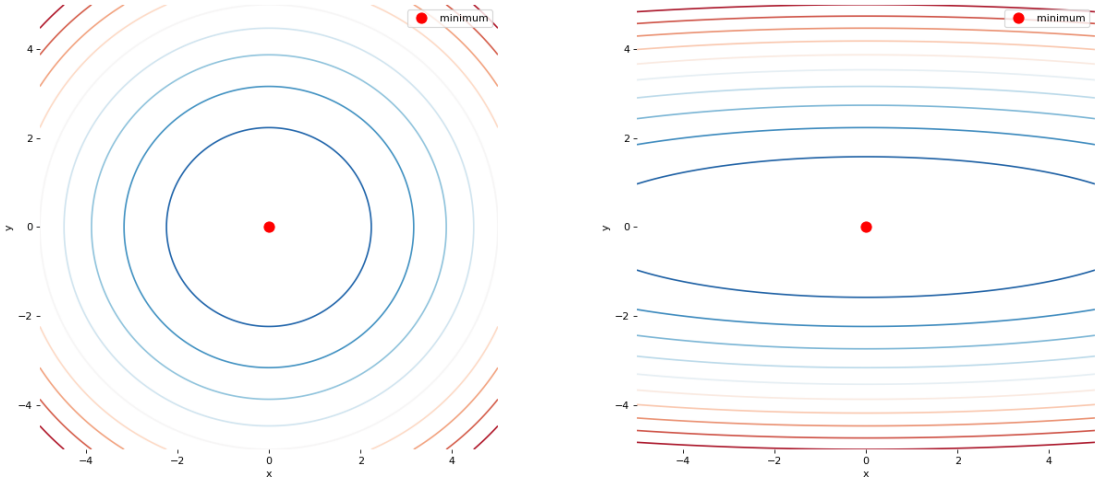




Adaptive learning rates

Adaptive learning rates

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



Isotropic vs. Anistropic

AdaGrad

Introduced in [1]. Per-parameter downscale of the learning rates by the square-root of the sum of squares of all the gradient historical values.

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

- Mitigates the need to manually tune the learning rate. Most implementations use $\gamma = 0.01$ as default
- Dynamic learning rates on each coordinate, learning rates can have different orders of magnitude across layers
- Accumulation of gradients acts as a descreasing learning rate
- r_t grows unboundedly during training, which may cause the step size to shrink and eventually become infinitesimally small
- Sensitive to initial condition: large initial gradients lead to small learning rates

---

[1]: Duchi, Hazan, and Singer, "Adaptive subgradient methods for online learning and stochastic optimization", Journal of machine learning research (2011)

RMSProp

Unpublished method, from the course of Geoff Hinton [1]

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

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

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

[1]: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf

AdaDelta

Introduced as an improvement of AdaGrad in [1]: less sensitivity to initial parameters.

$$
\begin{aligned}
r_t &= \rho r_{t-1} + (1-\rho) g_t \odot g_t \\
\theta_{t+1} &= \theta_t - \gamma \frac{\sqrt{\delta + s_{t-1}}}{\delta + \sqrt{r_t}} \odot g_t \\
s_t &= \rho s_{t-1} + (1-\rho)(\theta_{t+1} - \theta_t) \odot (\theta_{t+1} - \theta_t)
\end{aligned}
$$

- Actually uses mobile means using fixed number of past iterations to compute r_t and s_t
- The numerator keeps the size of the previous step in memory and enforce larger steps along directions in which large steps were made.
- The denominator keeps the size of the previous gradients in memory and acts as a decreasing learning rate.

---

[1]: Zeiler, "Adadelta: an adaptive learning rate method", arXiv preprint arXiv:1212.5701 (2012)

AdaDelta

The intuition is to compute a dynamic learning rate for each weight based on the past gradient values (as with AdaGrad) but using also second order information "a la" second order method.

Indeed, in second order methods we use

$$\theta_{t+1} - \theta_t = \left(\nabla^2 f(\theta_t)\right)^{-1} \nabla f(\theta_t)$$

so that, roughly,

$$\left(\nabla^2 f(\theta_t)\right)^{-1} \approx \frac{\theta_{t+1} - \theta_t}{\nabla f(\theta_t)}$$

see also [1]. But, citing [2]:

> Determining a good learning rate becomes more of an art than science for many problems

---

[1]: Schaul, Zhang and LeCun, "No more pesky learning rates", ICML (2013)
[2]: Zeiler, "Adadelta: an adaptive learning rate method", arXiv preprint arXiv:1212.5701 (2012)

Adam

Introduced in [1]. Similar to RMSProp with momentum, but uses first and second moment of the gradient to update the parameters.

$$
\begin{aligned}
s_t &= \rho_1 s_{t-1} + (1-\rho_1) g_t \\
\hat{s}_t &= \frac{s_t}{1-\rho_1^t} \\
r_t &= \rho_2 r_{t-1} + (1-\rho_2) g_t \odot g_t \\
\hat{r}_t &= \frac{r_t}{1-\rho_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\gamma}{\delta + \sqrt{\hat{r}_t}} \odot \hat{s}_t
\end{aligned}
$$

- Good defaults are $\delta = 10^{-3}, \rho_1 = 0.9, \rho_2 = 0.999$ and $\delta = 10^{-8}$
- Converge results are available in [1] and [2]
- Adam is one of the default optimizers in deep learning, along with SGD with momentum

---

[1]: Kingma and Ba, "Adam: A method for stochastic optimization", arXiv preprint (2014)
[2]: Reddi et al. "Adaptive methods for nonconvex optimization", NeurIPS (2018) ISO 690

Adamax

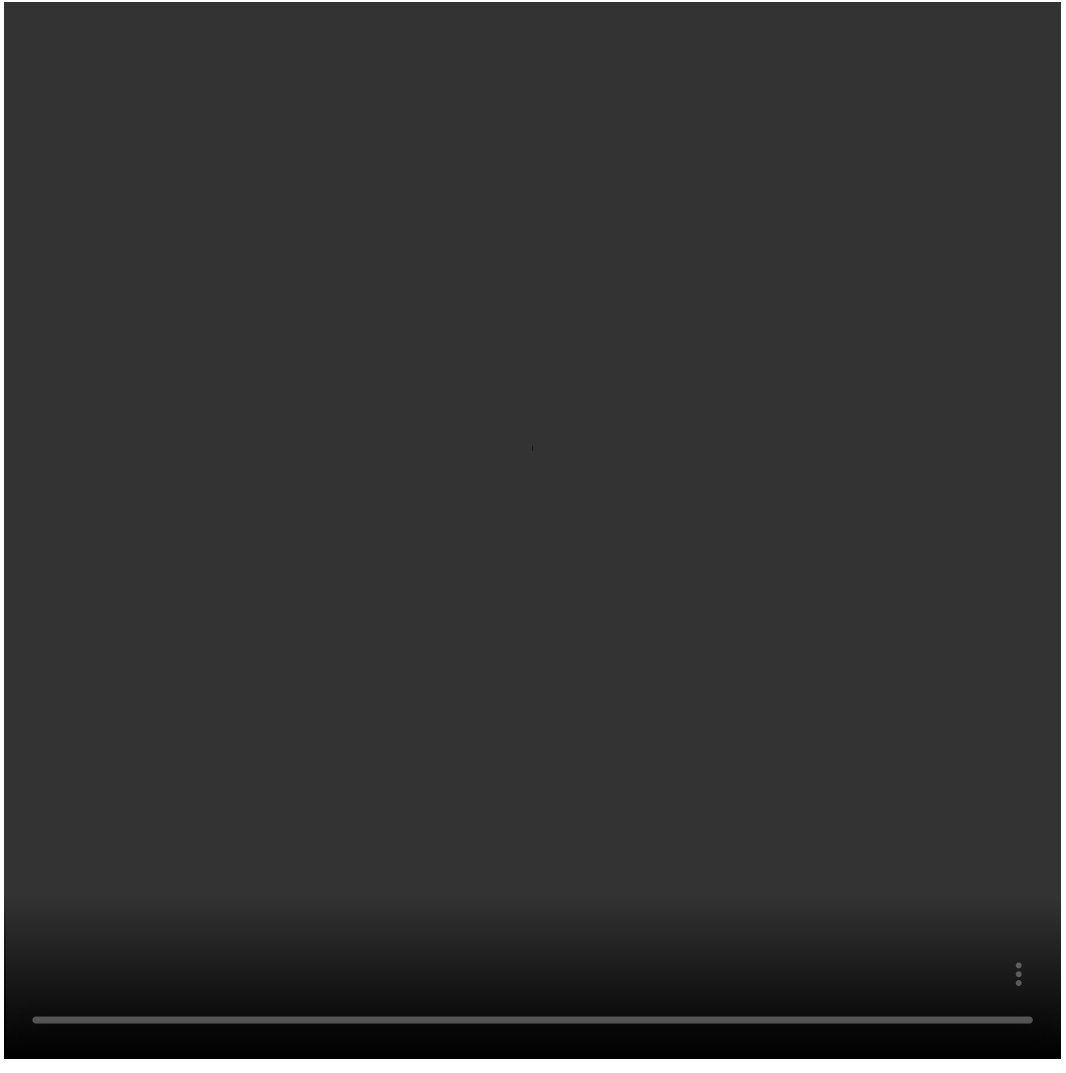
A variation on Adam introduced in [1]. Replaces the use of second-order moments by a exponentially decaying maximum of the absolute values of past gradients

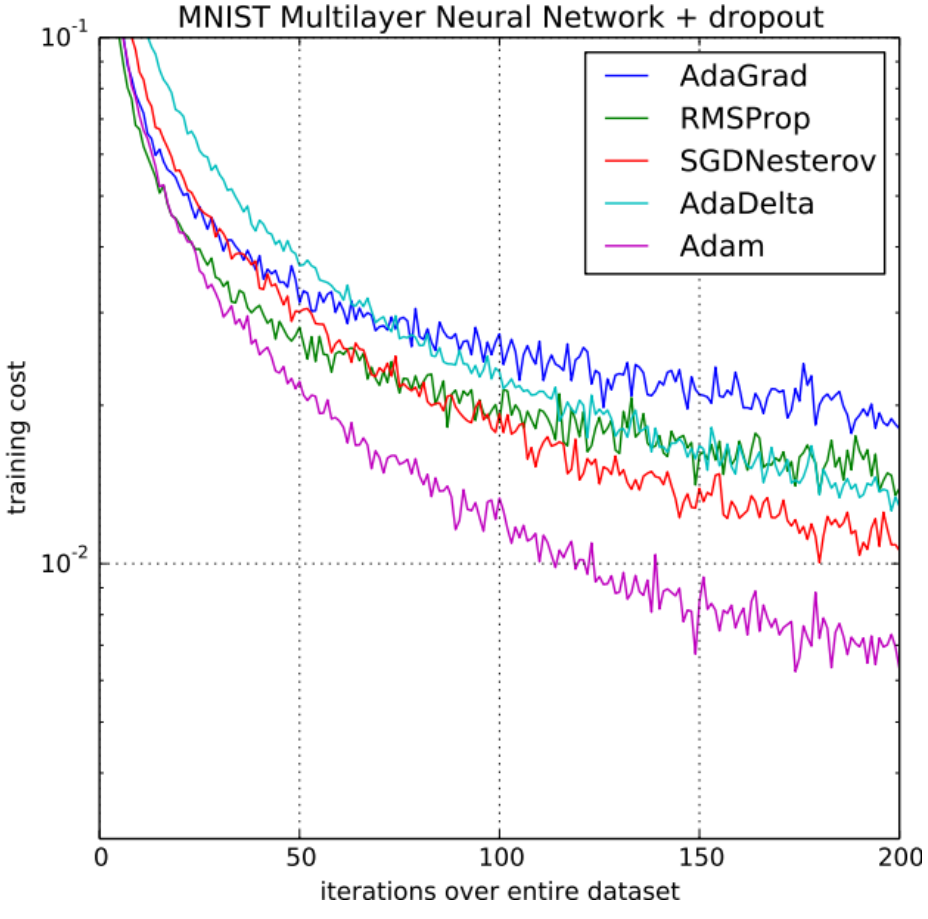
$$
\begin{aligned}
s_t &= \rho_1 s_{t-1} + (1 - \rho_1) g_t \\
\hat{s}_t &= \frac{s_t}{1 - \rho_1^t} \\
u_t &= \max\left(\rho_2 u_{t-1}, |g_t|\right) \\
\hat{r}_t &= \frac{r_t}{1 - \rho_2^t} \\
\theta_{t+1} &= \theta_t - \frac{\gamma}{\delta + u_t} \odot \hat{s}_t
\end{aligned}
$$

- Computationally cheaper, can lead to similar results as Adam

---

[1]: Kingma and Ba, "Adam: A method for stochastic optimization", arXiv preprint (2014)





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

Learning rate scheduling

Learning rate scheduling

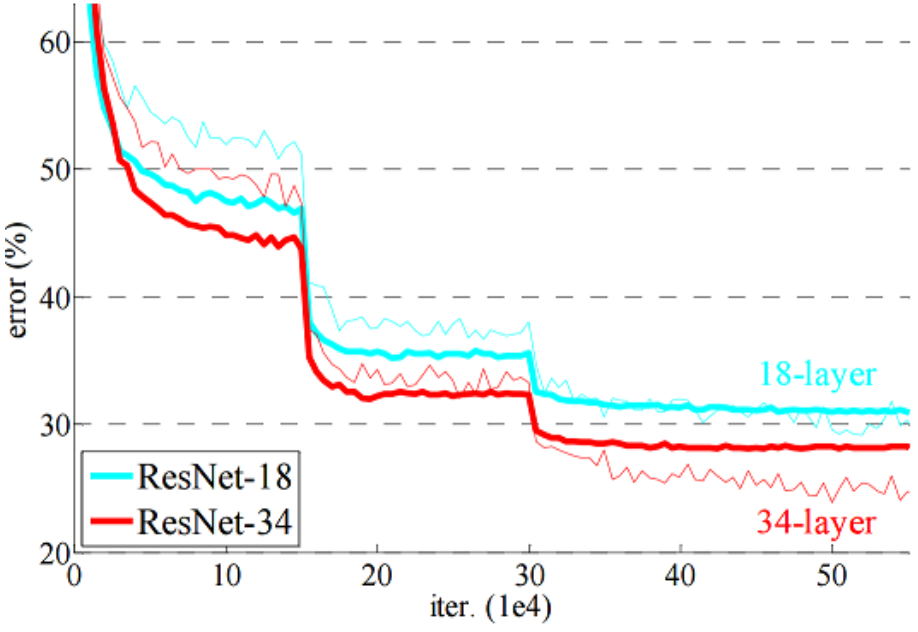
Despite per-parameter adaptive learning rate methods, it is usually helpful to anneal the learning rate γ over time.

- 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.

You will also find other techniques such as:

- Warm-up phases, where the learning rate is increased before being annealed
- Oscillating learning rates

Learning rate scheduling



Step decay scheduling for training ResNets.

Thank you !