Lecture 5 - Training & Regularizing Neural Networks

DD2424, Josephine Sullivan

March 26, 2025

Today's Lecture

- **Improving the training error**
  - Improving on vanilla mini-batch gradient descent
  - Learning rate schedulers
- **Improving the test error**
  - Regularization
  - Choosing hyper-parameters

Optimization: Improving on Stochastic Gradient Descent/Gradient Descent

Remember Gradient Descent optimization

- Want to minimize the function $f : \mathbb{R}^d \to \mathbb{R}$
- Gradient descent update step at time t

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \eta_t \nabla f(\mathbf{x}^{(t)})$$

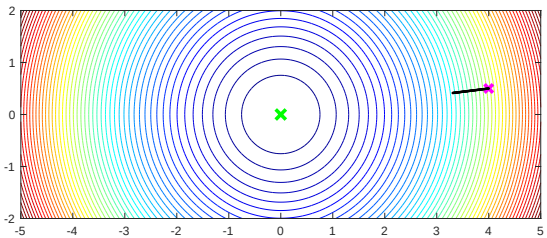
- How to choose η_t ?
- Depends on
  - the shape of f and
  - how fast a convergence you want.
- **Important property**:
  Gradient vector $\nabla f(\mathbf{x}^{(t)})$ is always orthogonal to the iso-contour curve of f at $\mathbf{x}^{(t)}$.

Example behaviour on the ideal f for gradient descent

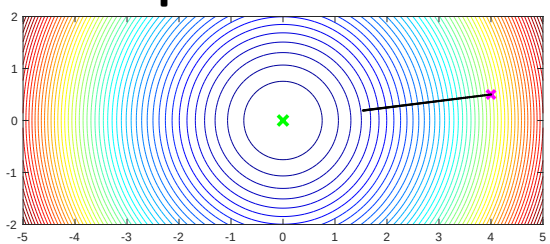
Use **gradient descent** to minimize $f(\mathbf{x}) = x_1^2 + x_2^2$ with

- $\mathbf{x}_0 = (4, .5)^T$,
- η_t constant at each iteration and
- a fixed number of update steps.

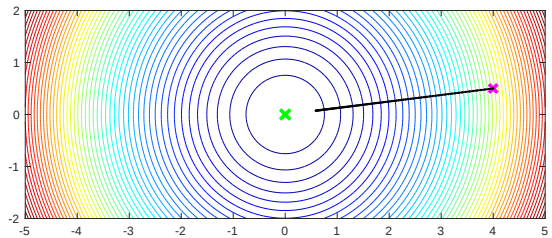
Gradient descent paths for different η_t's



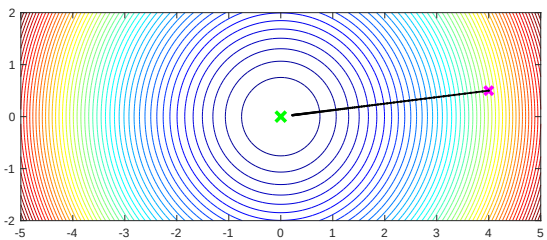
$\eta_t = .001$



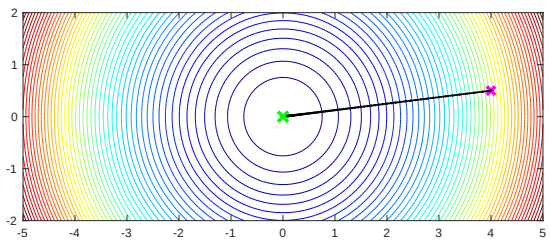
$\eta_t = .005$



$\eta_t = .01$



$\eta_t = .015$



$\eta_t = .02$

Note in this example the update step always points directly to the optimum point

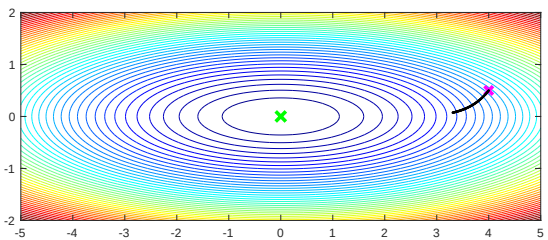
$\Longrightarrow$ can increase η_t liberally (without worrying about divergence) to speed up convergence.

Example behaviour on a suitable f for gradient descent

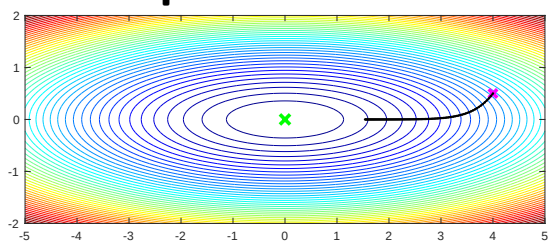
Use gradient descent to minimize $f(\mathbf{x}) = x_1^2 + 10x_2^2$ with

- $\mathbf{x}_0 = (4, .5)^T$,
- η_t constant at each iteration and
- a fixed number of update steps.

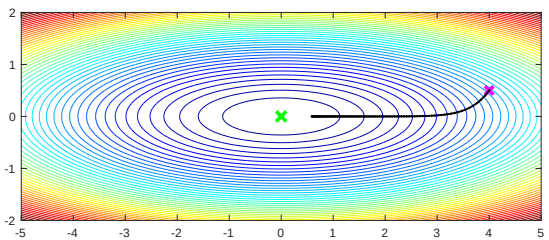
Gradient descent paths for different η_t's



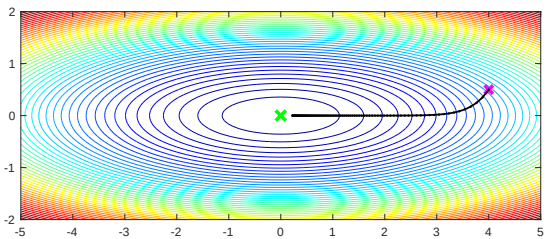
$\eta_t = .001$



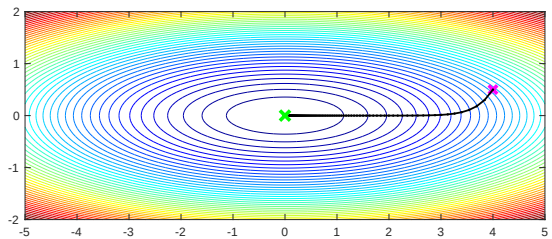
$\eta_t = .005$



$\eta_t = .01$



$\eta_t = .015$



$\eta_t = .02$

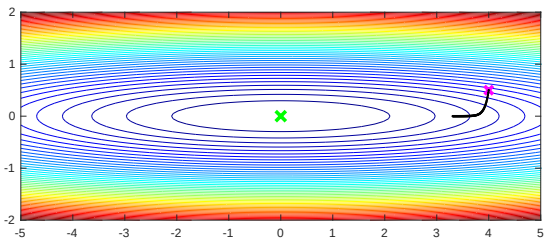
Note in this example the update direction points directly to the optimum when $x_{t,2} = 0$.

Example behaviour on a less suitable f for gradient descent

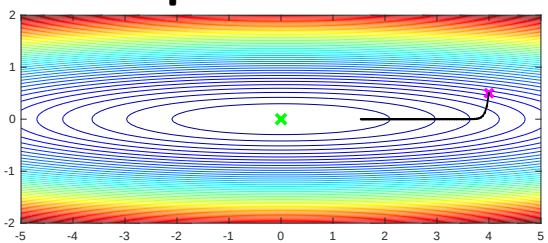
Use gradient descent to minimize $f(\mathbf{x}) = x_1^2 + 50x_2^2$ with

- $\mathbf{x}_0 = (4, .5)^T$,
- η_t constant at each iteration and
- a fixed number of update steps.

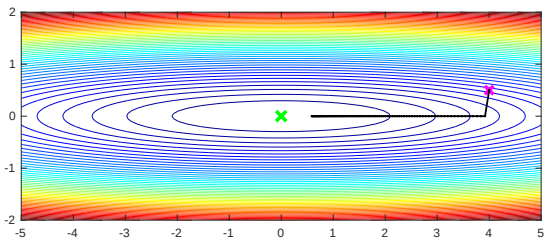
Gradient descent path for different η_t's



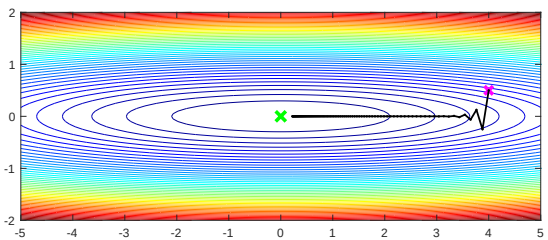
$\eta_t = .001$



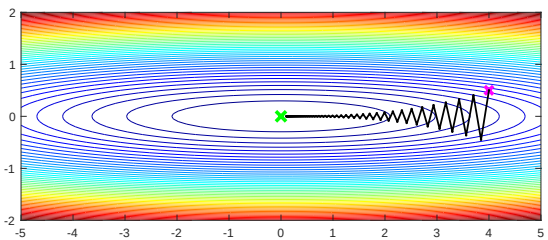
$\eta_t = .005$



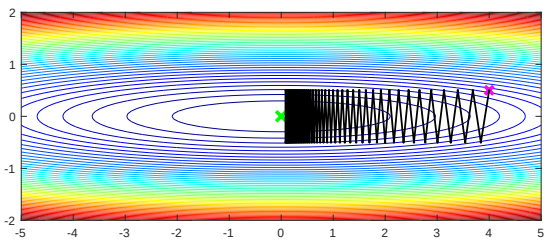
$\eta_t = .01$



$\eta_t = .015$



$\eta_t = .019$



$\eta_t = .02$

Note as η_t increases GD path increasingly zig-zags across the valley. In this case GD diverges for $\eta_t > .02$. To begin all paths move $\approx$ orthogonal to *optimum* direction.

Focus on GD and simple quadratic cost function

The set-up

Want to minimize w.r.t. $\mathbf{x}$:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} \quad \text{where } A = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

and $\lambda_1 > 0$ and $\lambda_2 > 0$.
(The optimum value is at $\mathbf{x}^* = \mathbf{0}$.)

Want to minimize w.r.t. $\mathbf{x}$:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} \quad \text{where } A = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

and $\lambda_1 > 0$ and $\lambda_2 > 0$.
(The optimum value is at $\mathbf{x}^* = \mathbf{0}$.)

How does GD work on this problem?

- Apply GD with fixed η and initial guess $\mathbf{x}^{(0)}$ for $\mathbf{x}^*$
- Questions:
  - For what values of η will GD converge?
  - What is the rate of convergence?

The GD estimate at update t

Quadratic function we want to minimize:

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T A\mathbf{x} \quad \text{where } A = \begin{pmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{pmatrix}$$

and $\lambda_1 > 0$ and $\lambda_2 > 0$.

- The gradient $\nabla_{\mathbf{x}} f(\mathbf{x})$ is

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = A\mathbf{x}$$

- At update t have

$$\begin{aligned} \mathbf{x}^{(t)} &= (I - \eta A)^t \, \mathbf{x}^{(0)} \\ &= \begin{pmatrix} (1-\eta\lambda_1)^t & 0 \\ 0 & (1-\eta\lambda_2)^t \end{pmatrix} \mathbf{x}^{(0)} \end{aligned}$$

Convergence when?

- At update t have

$$\mathbf{x}^{(t)} = \begin{pmatrix} (1-\eta\lambda_1)^t & 0 \\ 0 & (1-\eta\lambda_2)^t \end{pmatrix} \mathbf{x}^{(0)}$$

- For convergence ($\mathbf{x}^{(t)} \to \mathbf{x}^*$ at $t \to \infty$) need

$$|1-\eta\lambda_1| < 1 \quad \text{and} \quad |1-\eta\lambda_2| < 1$$

$$\Longrightarrow$$

$$0 < \eta < \frac{2}{\max(\lambda_1, \lambda_2)}$$

η that gives fastest convergence?

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2} (1 - \eta\lambda_i)^{2t} \lambda_i \, {x_i^{(0)}}^2$$

- The **rate of convergence** of above to zero is determined by

$$\max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Want to find the $0 < \eta < 2/\max(\lambda_1, \lambda_2)$ that maximizes the rate of convergence $\implies$

$$\min_{\eta} \max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Solution is

$$\eta^* = \frac{2}{\lambda_1 + \lambda_2}$$

η that gives fastest convergence?

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2} (1 - \eta\lambda_i)^{2t} \lambda_i \, {x_i^{(0)}}^2$$

- The **rate of convergence** of above to zero is determined by

$$\max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Want to find the $0 < \eta < 2/\max(\lambda_1, \lambda_2)$ that maximizes the rate of convergence $\implies$

$$\min_{\eta} \max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Solution is

$$\eta^* = \frac{2}{\lambda_1 + \lambda_2}$$

η that gives fastest convergence?

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2} (1 - \eta\lambda_i)^{2t} \lambda_i \, {x_i^{(0)}}^2$$

- The **rate of convergence** of above to zero is determined by

$$\max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Want to find the $0 < \eta < 2/\max(\lambda_1, \lambda_2)$ that maximizes the rate of convergence $\implies$

$$\min_{\eta} \; \max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Solution is

$$\eta^* = \frac{2}{\lambda_1 + \lambda_2}$$

η that gives fastest convergence?

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2} (1 - \eta\lambda_i)^{2t} \lambda_i \, {x_i^{(0)}}^2$$

- The **rate of convergence** of above to zero is determined by

$$\max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Want to find the $0 < \eta < 2/\max(\lambda_1, \lambda_2)$ that maximizes the rate of convergence $\implies$

$$\min_{\eta} \max(|1 - \eta\lambda_1|, |1 - \eta\lambda_2|)$$

- Solution is

$$\eta^* = \frac{2}{\lambda_1 + \lambda_2}$$

Rate of convergence at η^*

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2}(1-\eta\lambda_i)^{2t}\lambda_i \, {x_i^{(0)}}^2$$

- For the optimal learning rate have

$$|1-\eta^*\lambda_1| = |1-\eta^*\lambda_2| = \frac{\lambda_{\max}/\lambda_{\min}-1}{\lambda_{\max}/\lambda_{\min}+1}$$

- Let $\kappa = \lambda_{\max}/\lambda_{\min}$ then rate of convergence is defined by

$$\frac{\kappa-1}{\kappa+1}$$

- Larger $\kappa \implies$ closer $(\kappa-1)/(\kappa+1)$ is to $1 \implies$ slower the convergence of the GD estimate.

Rate of convergence at η^*

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2}(1-\eta\lambda_i)^{2t}\lambda_i\, {x_i^{(0)}}^2$$

- For the optimal learning rate have

$$|1-\eta^*\lambda_1| = |1-\eta^*\lambda_2| = \frac{\lambda_{\max}/\lambda_{\min}-1}{\lambda_{\max}/\lambda_{\min}+1}$$

- Let $\kappa = \lambda_{\max}/\lambda_{\min}$ then rate of convergence is defined by

$$\frac{\kappa-1}{\kappa+1}$$

- Larger $\kappa \implies$ closer $(\kappa-1)/(\kappa+1)$ is to $1 \implies$ slower the convergence of the GD estimate.

Rate of convergence at η^*

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2}(1-\eta\lambda_i)^{2t}\lambda_i\, {x_i^{(0)}}^2$$

- For the optimal learning rate have

$$|1-\eta^*\lambda_1| = |1-\eta^*\lambda_2| = \frac{\lambda_{\max}/\lambda_{\min}-1}{\lambda_{\max}/\lambda_{\min}+1}$$

- Let $\kappa = \lambda_{\max}/\lambda_{\min}$ then rate of convergence is defined by

$$\frac{\kappa-1}{\kappa+1}$$

- Larger $\kappa \implies$ closer $(\kappa-1)/(\kappa+1)$ is to $1 \implies$ slower the convergence of the GD estimate.

Rate of convergence at η^*

- Difference between function evaluated at $\mathbf{x}^{(t)}$ and $\mathbf{x}^*$:

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{2} (1 - \eta\lambda_i)^{2t} \lambda_i \, {x_i^{(0)}}^2$$

- For the optimal learning rate have

$$|1 - \eta^*\lambda_1| = |1 - \eta^*\lambda_2| = \frac{\lambda_{\max}/\lambda_{\min} - 1}{\lambda_{\max}/\lambda_{\min} + 1}$$

- Let $\kappa = \lambda_{\max}/\lambda_{\min}$ then rate of convergence is defined by

$$\frac{\kappa - 1}{\kappa + 1}$$

- Larger $\kappa \implies$ closer $(\kappa - 1)/(\kappa + 1)$ is to $1 \implies$ slower the convergence of the GD estimate.

Rate of convergence depends on ratio - $\lambda_{\max}/\lambda_{\min}$

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\|^2 = \sum_{i=1}^{2} \left(\frac{\kappa - 1}{\kappa + 1}\right)^{2t} \lambda_i \, {x_i^{(0)}}^2$$

where

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$$

- κ is known as the condition number (indicating how close a matrix is to singular)
- When $\kappa = 1$ for η^* have convergence in one update step
- When $\kappa \gg 1$ for η^* (and all other valid η) have poor convergence rate for GD.

Can extend result to d dimensions

Have

$$A = \text{diag}(\lambda_1, \ldots, \lambda_d)$$

- GD converges when

$$0 < \eta < \frac{2}{\lambda_{\max}}$$

- Have fastest convergence when

$$\eta^* = \frac{2}{\lambda_{\max} + \lambda_{\min}}$$

- Then for optimal η^*

$$\|f(\mathbf{x}^{(t)}) - f(\mathbf{x}^*)\| = \sum_{i=1}^{d} \left(\frac{\kappa - 1}{\kappa + 1}\right)^{2t} \lambda_i \, {x_i^{(0)}}^2$$

  where $\kappa = \lambda_{\max}/\lambda_{\min}$.

- When κ increases convergence rate of GD decreases.

Back to our discussion...

Pathological case for GD optimization

- The curvature in different directions is very different for f.
- Optimization results in a zig-zagging path across the ravine $\Longrightarrow$ slow convergence.

Unfortunately, in neural network cost functions

ravines are common around local optima.

Gradient descent and the problem of choosing η_t

- η_t too small $\implies$ slow convergence speed but smooth update path.
- η_t too large $\implies$ optimization can potentially diverge and potentially have an inefficient zig-zag update path.

Review & summary problems with SGD

- Local minima - Many, but hopefully can avoid bad ones with reasonable initialization.
- Saddle points - zero gradient, gradient descent gets stuck & lots of these on the loss surface for neural networks
- Poor conditioning - hard to get to local minimum because of ravines
- Gradients come from mini-batches so they can be noisy!

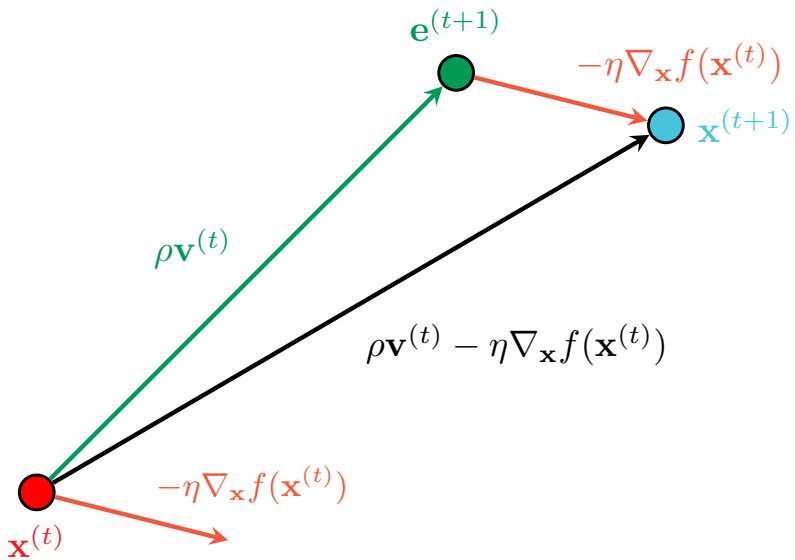
Improvement 1: SGD with momentum

- Introduce **momentum** vector as well as the gradient vector.
  Can think of as a running mean of gradients

- Let $\rho \in [0, 1]$ and $\mathbf{v}$ is the momentum vector

$$\mathbf{v}^{(t+1)} = \rho\,\mathbf{v}^{(t)} - \eta \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)}) \quad \leftarrow \text{update vector}$$
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \mathbf{v}^{(t+1)}$$

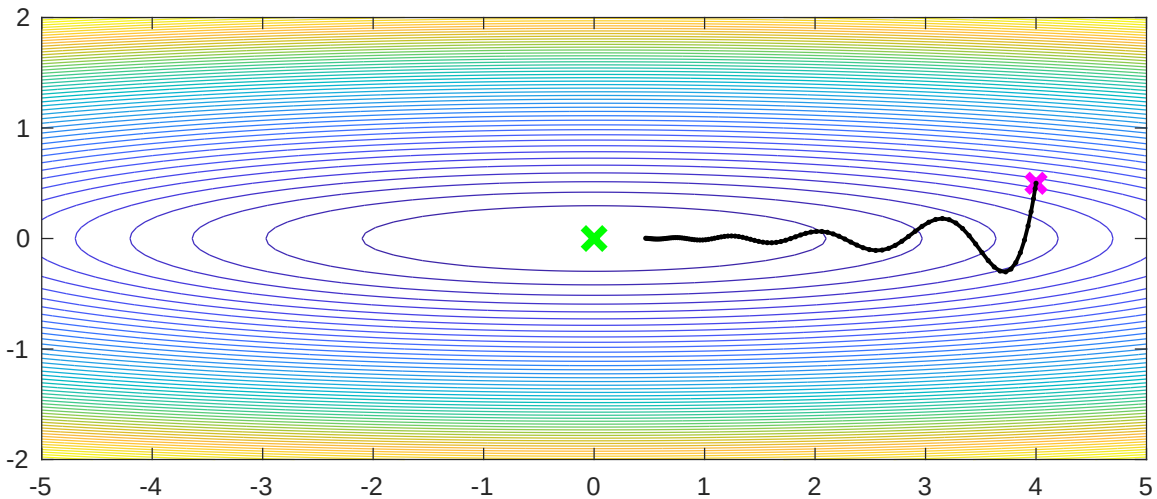
  Typically ρ set to .9 or .99.



How and why momentum helps

How?

- Momentum helps accelerate SGD in the appropriate direction.
- Momentum dampens the oscillations of default SGD.
  $\implies$ **Faster convergence**.



($\rho = .9$, $\eta = .001$, 95 update steps)

Why?

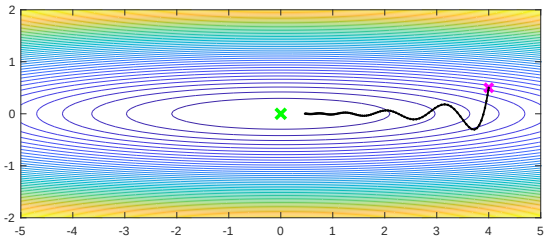
- For dimensions whose gradient is constantly changing then their entries in the update vector are damped.
- For dimensions whose gradient is **approx. constant** then their entries in the update vector are **not damped**.

Example behaviour of "GD + mom" for a ravine

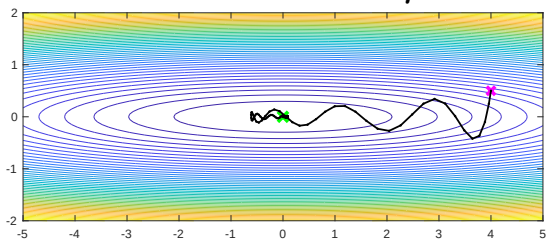
Use gradient descent to minimize $f(\mathbf{x}) = x_1^2 + 50x_2^2$ with

- $\mathbf{x}_0 = (4, .5)^T$,
- η_t constant at each iteration and
- a fixed number of update steps.

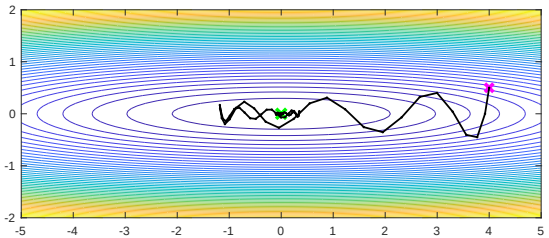
GD + mom path for different η_t's and $\rho = .9$



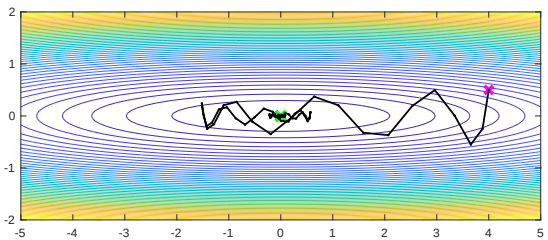
$\eta_t = .001$



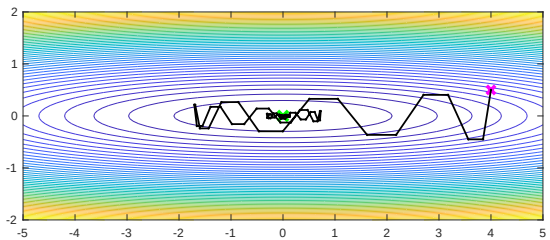
$\eta_t = .005$



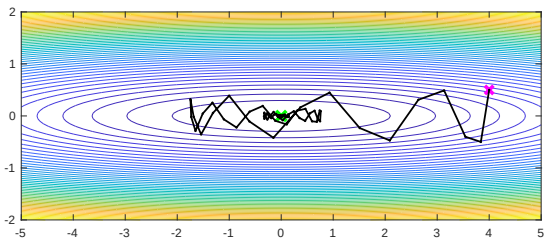
$\eta_t = .01$



$\eta_t = .015$



$\eta_t = .019$



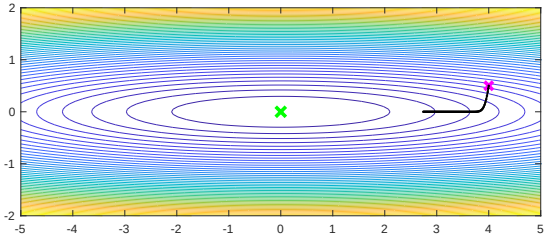
$\eta_t = .02$

Example behaviour of "GD + mom" for a ravine

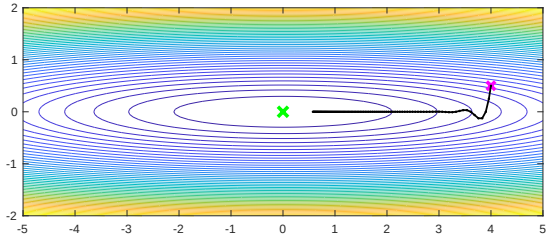
Use gradient descent to minimize $f(\mathbf{x}) = x_1^2 + 50x_2^2$ with

- $\mathbf{x}_0 = (4, .5)^T$,
- η_t constant at each iteration and
- a fixed number of update steps.

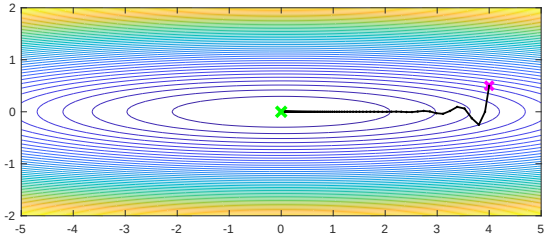
GD + mom path for different η_t's and $\rho = .5$



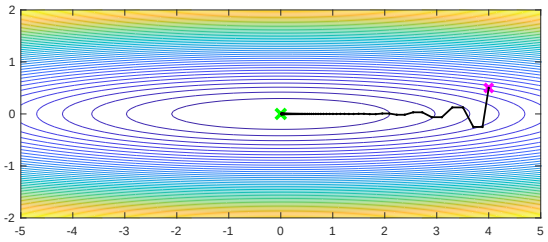
$\eta_t = .001$



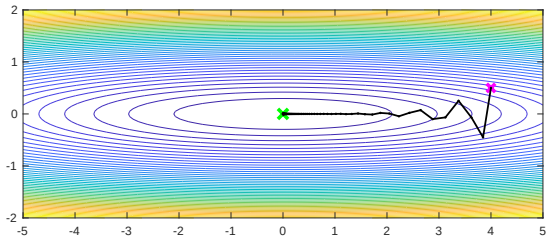
$\eta_t = .005$



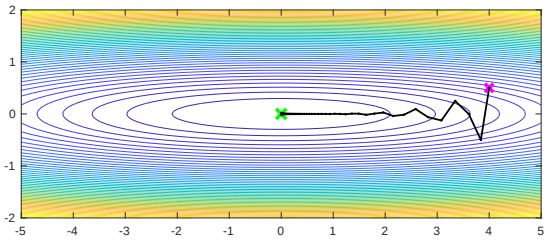
$\eta_t = .01$



$\eta_t = .015$



$\eta_t = .019$



$\eta_t = .02$

Have only scratched the surface here...

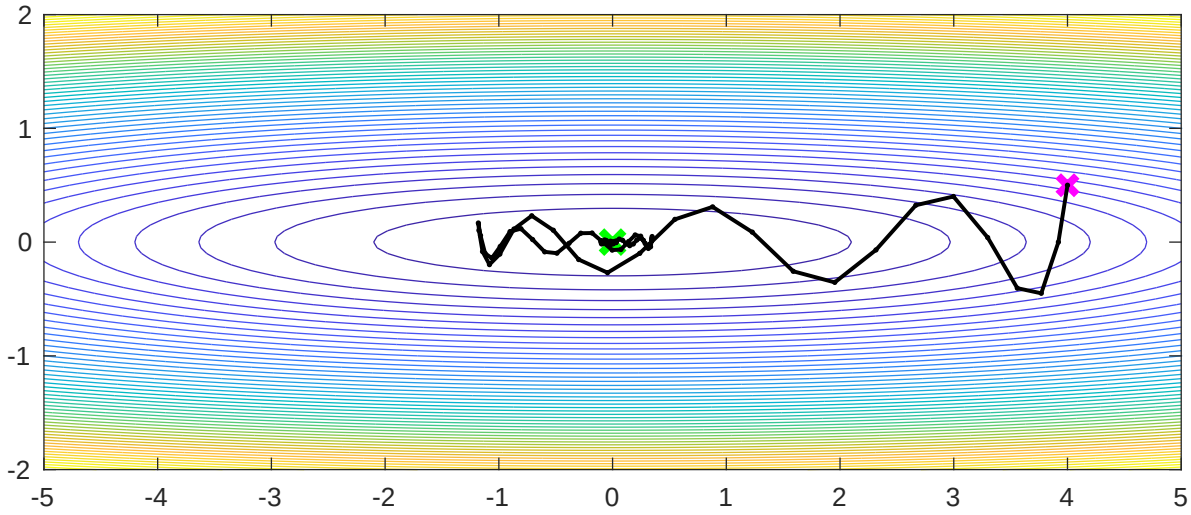
For a more sophisticated analysis and clear exposition of momentum please check out: Why Momentum Really Works

Momentum not the complete answer

- When using momentum

  $\Longrightarrow$ can pick up too much speed in one direction.

  $\Longrightarrow$ can overshoot the local optimum.



($\rho = .9$, $\eta = .01$, 95 update steps)

Solution: Nesterov accelerated gradient (NAG)

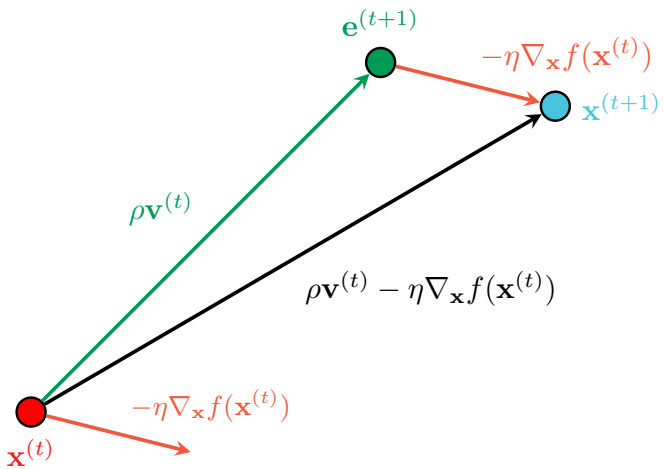
- Look and measure ahead.
- Use gradient at an estimate of the parameters at the next iteration.
- Let $\gamma \in [0, 1]$ then

$$\mathbf{e}^{(t+1)} = \mathbf{x}^{(t)} + \rho \mathbf{v}^{(t)} \quad \leftarrow \text{estimate of } \mathbf{x}^{(t+1)}$$

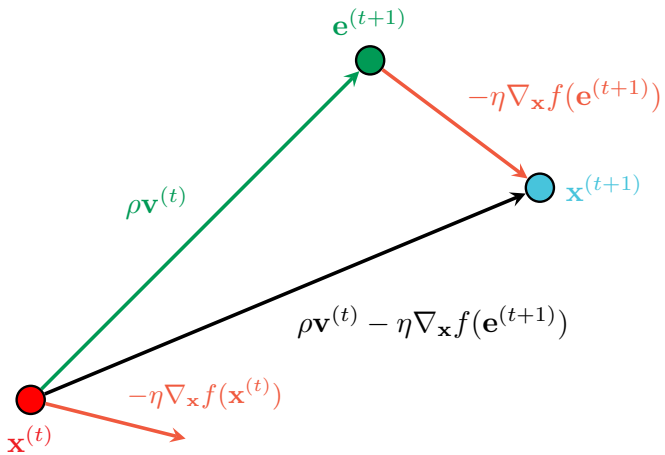
$$\mathbf{v}^{(t+1)} = \rho\, \mathbf{v}^{(t)} - \eta\, \nabla_{\mathbf{x}} f(\mathbf{e}^{(t+1)}) \quad \leftarrow \text{update vector}$$

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \mathbf{v}^{(t+1)}$$

Typically ρ set to .9.



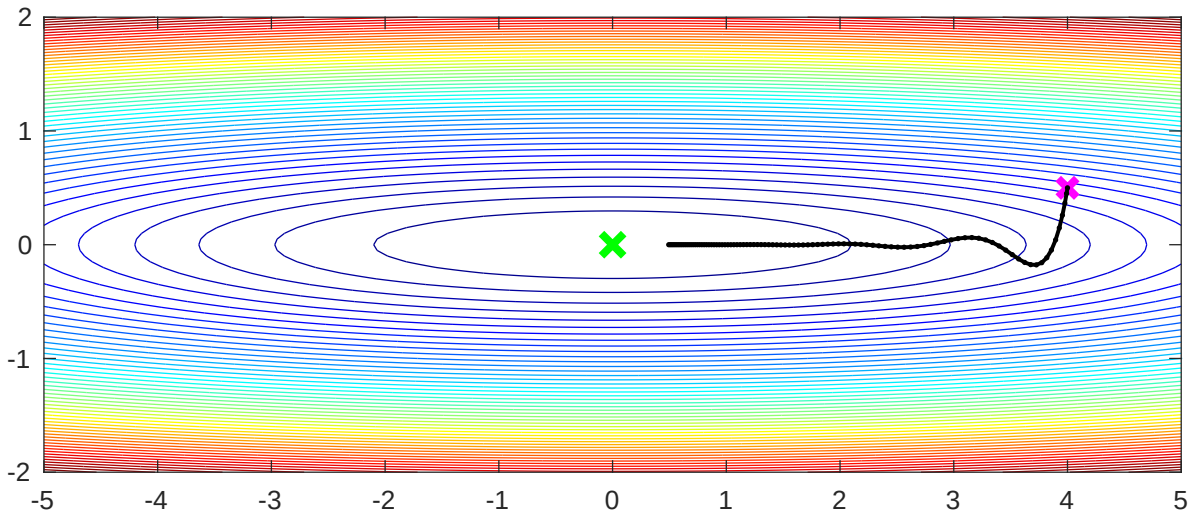
Momentum update



NAG update

How and why NAG helps

- The anticipatory update prevents the algorithm having too large updates and overshooting.
- Algorithm has increased responsiveness to the landscape of f.



($\rho = .9$, $\eta = .001$, 95 update steps)

Note: NAG shown to greatly increase the ability to train RNNs:

Bengio, Y., Boulanger-Lewandowski, N. & Pascanu, R. *Advances in Optimizing Recurrent Networks, (2012).*

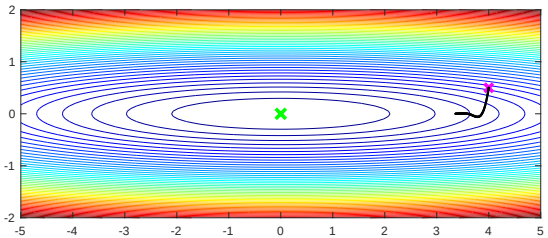
`http://arxiv.org/abs/1212.0901`

Example behaviour of "NAG" for a ravine

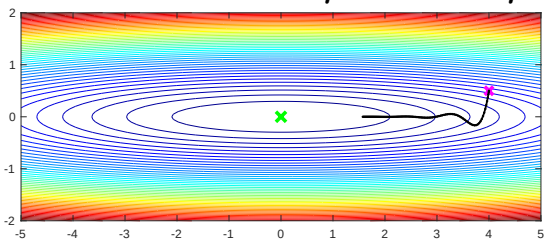
Use gradient descent to minimize $f(\mathbf{x}) = x_1^2 + 50x_2^2$ with

- $\mathbf{x}_0 = (4, .5)^T$,
- η_t constant at each iteration and
- a fixed number of update steps.

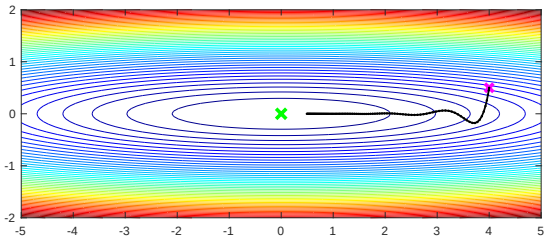
NAG path for different η_t's and $\rho = .9$



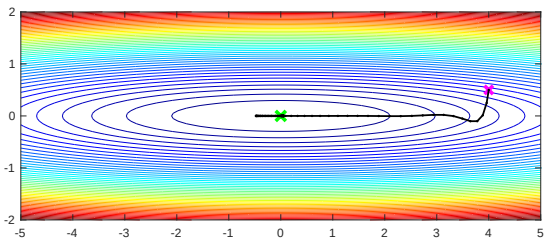
$\eta_t = .0001$



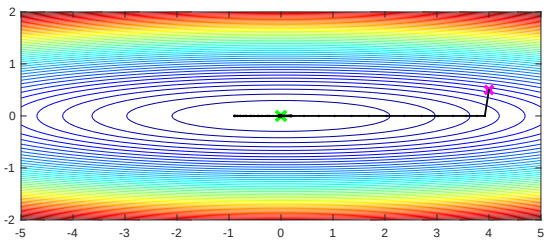
$\eta_t = .0005$



$\eta_t = .001$



$\eta_t = .005$



$\eta_t = .01$

Re-define NAG

Slightly annoying formulation of NAG

$$\mathbf{v}^{(t+1)} = \rho\,\mathbf{v}^{(t)} - \eta\,\nabla_{\mathbf{x}} f(\mathbf{x}^{(t)} + \rho\mathbf{v}^{(t)})$$
$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} + \mathbf{v}^{(t+1)}$$

Would prefer updates in terms of $\mathbf{x}^{(t)}$ and $\nabla_{\mathbf{x}} f(\mathbf{x}^{(t)})$!

- To achieve this have a change of variables

$$\tilde{\mathbf{x}}^{(t)} = \mathbf{x}^{(t+1)} + \rho\mathbf{v}^{(t+1)}$$

and re-arrange terms to get

$$\mathbf{v}^{(t+1)} = \rho\,\mathbf{v}^{(t)} - \eta\,\nabla_{\mathbf{x}} f(\tilde{\mathbf{x}}^{(t)}) \quad \leftarrow \text{update vector}$$
$$\tilde{\mathbf{x}}^{(t+1)} = \tilde{\mathbf{x}}^{(t)} + \mathbf{v}^{(t+1)} + \rho\left(\mathbf{v}^{(t+1)} - \mathbf{v}^{(t)}\right)$$

Improvements to NAG

- Want to **adapt the updates** to each **individual parameter**.
- Perform larger or smaller updates depending on the landscape of the cost function.
- Family of algorithms with **adaptive learning rates**
  - AdaGrad
  - AdaDelta
  - RMSProp
  - Adam

AdaGrad

- **Idea**: Downscale a model parameter by square-root of sum of squares of all its historical values.

- Parameters that have large partial derivative of the loss – learning rates for them are rapidly declined

AdaGrad

- For a cleaner statement introduce some notation:

$$\mathbf{g}_t = \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)}) \quad \text{and } \mathbf{g}_t = (g_{t,1}, \ldots, g_{t,d})^T.$$

- Keep a record of the sum of the squares of the gradients w.r.t. each x_i up to time t:

$$G_{t,i} = \sum_{j=1}^{t} g_{j,i}^2$$

- The AdaGrad update step for each dimension is

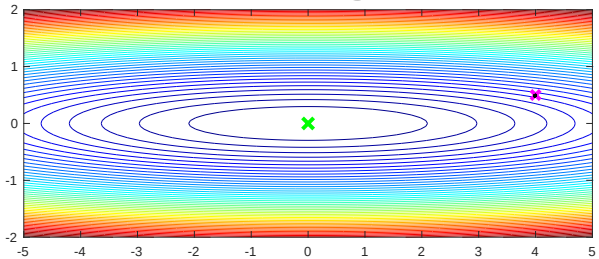
$$x_i^{(t+1)} = x_i^{(t)} - \frac{\eta}{\sqrt{G_{t,i} + \epsilon}} g_{t,i}$$

- Usually set $\epsilon = 1e-8$ and $\eta = .01$.

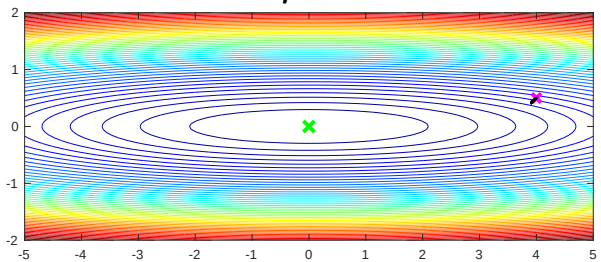
J. Duchi, E. Hazan & Y. Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research, 2011.

Adagrad's performance on the ravine

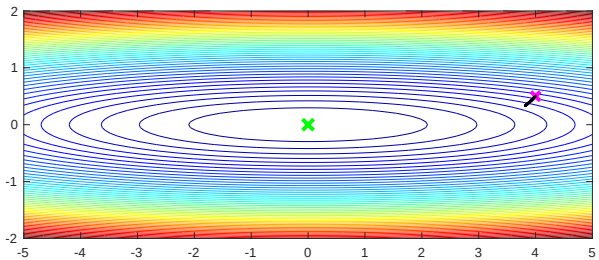
Adagrad's paths for different η_t's



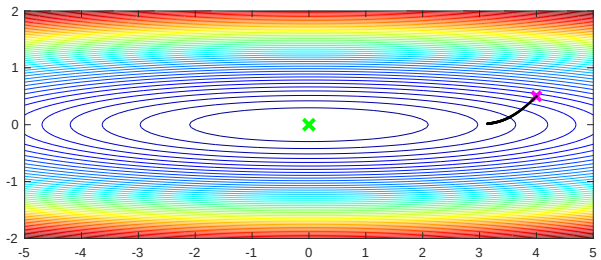
$\eta_t = .001$



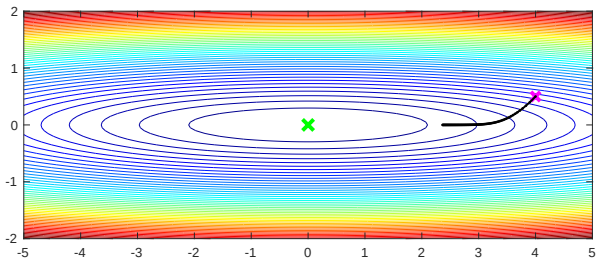
$\eta_t = .005$



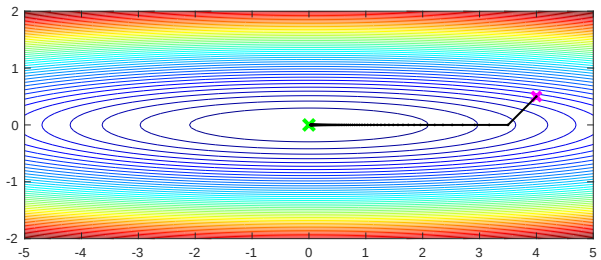
$\eta_t = .01$



$\eta_t = .05$



$\eta_t = .1$



$\eta_t = .5$

Progress along "steep" directions is damped; Progress along "flat" directions is accelerated.

Big weakness of AdaGrad

- Each $g_{t,i}^2$ is positive. At t increases

  $\Longrightarrow$ Each $G_{t,i} = \sum_{j=1}^{t} g_{j,i}^2$ keeps growing during training.

  $\Longrightarrow$ the effective learning rate $\eta/(\sqrt{G_{t,i} + \epsilon})$ shrinks and eventually $\longrightarrow 0$.

  $\Longrightarrow$ updates of $\mathbf{x}^{(t)}$ stop for large t.

RMSProp

RMSProp addresses AdaGrad's radically diminishing learning rate:

- RMSProp an **adaptive learning rate** method proposed by Geoff Hinton in *Lecture 6e of his Coursera Class*.
- Stores an exponentially decaying average of the square of the gradient vector:

$$E\left[\mathbf{g}_{t+1}^2\right] = \gamma\ E\left[\mathbf{g}_t^2\right] + (1-\gamma)\,\mathbf{g}_{t+1}^2$$

- The RMSProp update rule:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{\eta}{\sqrt{E\left[\mathbf{g}_{t+1}^2\right] + \epsilon}}\,\mathbf{g}_{t+1}$$

- Typically set $\gamma = .9$ and $\eta = 0.001$.

AdaDelta

- Devised as an improvement to AdaGrad.
- Tackles AdaGrad's convergence to zero of the learning rate as t increases.
- AdaDelta's two central ideas
  - Scale learning rate based on the previous gradient values (like AdaGrad) but only using a recent time window,
  - Include an acceleration term (like momentum) by accumulating prior updates.

M. Zeiler, *ADADELTA: An Adaptive Learning Rate Method*, 2012. `http://arxiv.org/abs/1212.5701`

Technical details of AdaDelta

- Compute gradient vector $\mathbf{g}_t$ at current estimate $\mathbf{x}^{(t)}$.
- Update average of previous squared gradients (AdaGrad-like step)

$$\tilde{G}_{t,i} = \rho\, \tilde{G}_{t-1,i} + (1-\rho)\, g_{t,i}^2$$

- Compute exponentially decaying average of updates (momentum-like step)

$$U_{t,i} = \rho\, U_{t-1,i} + (1-\rho)\, u_{t,i}^2$$

- Compute the update vector

$$u_{t,i} = \frac{\sqrt{U_{t-1,i} + \epsilon}}{\sqrt{\tilde{G}_{t,i} + \epsilon}}\, g_{t,i}$$

- The AdaDelta update step:

$$x_i^{(t+1)} = x_i^{(t)} - u_{t,i}$$

Adaptive Moment Estimation (Adam)

- Computes **adaptive learning rates** for each parameter.
- How?
  - Stores an **exponentially decaying average** of
    - ⋆ past gradients $\mathbf{m}^{(t)}$ and
    - ⋆ past squared gradients $\mathbf{v}^{(t)}$
  - $\mathbf{m}^{(t)}$ and $\mathbf{v}^{(t)}$ estimate the mean and variance of the sequence of computed gradients in each dimension.
  - Uses the variance estimate to
    - ⋆ damp the update in dimensions varying alot and
    - ⋆ increase the update in dimensions with low variation.

D. P. Kingma & J. L. Ba, *Adam: a Method for Stochastic Optimization*, International Conference on Learning Representations, 2015.

Update equations for Adam

- Let $\mathbf{g}_t = \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)})$

$$\mathbf{m}^{(t+1)} = \beta_1 \mathbf{m}^{(t)} + (1 - \beta_1)\, \mathbf{g}_t$$
$$\mathbf{v}^{(t+1)} = \beta_2 \mathbf{v}^{(t)} + (1 - \beta_2)\, \mathbf{g}_t \;.*\; \mathbf{g}_t$$

- Set $\mathbf{m}^{(0)} = \mathbf{v}^{(0)} = \mathbf{0} \implies \mathbf{m}^{(t)}$ and $\mathbf{v}^{(t)}$ are biased towards zero (especially during the initial time-steps).
- Counter these biases by setting:

$$\hat{\mathbf{m}}^{(t+1)} = \frac{\mathbf{m}^{(t+1)}}{1 - \beta_1^t}, \qquad \hat{\mathbf{v}}^{(t+1)} = \frac{\mathbf{v}^{(t+1)}}{1 - \beta_2^t}$$

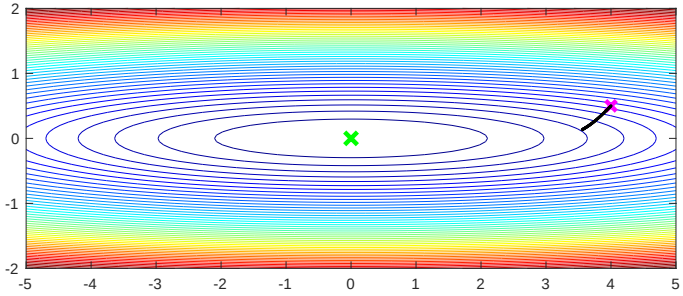
- The Adam update rule:

$$\mathbf{x}^{(t+1)} = \mathbf{x}^{(t)} - \frac{\eta}{\sqrt{\hat{\mathbf{v}}^{(t+1)}} + \epsilon}\, \hat{\mathbf{m}}^{(t+1)}$$

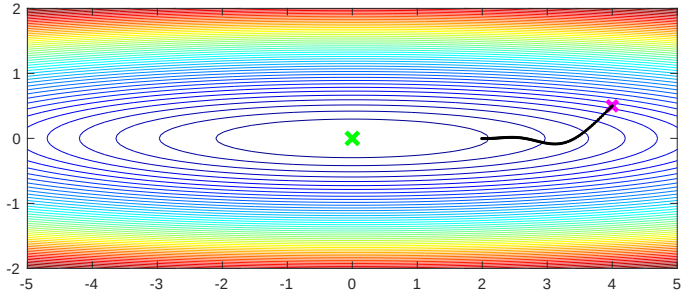
- Suggested default values $\beta_1 = .9, \beta_2 = .999, \epsilon = 10^{-8}$.

Adam's performance on the ravine

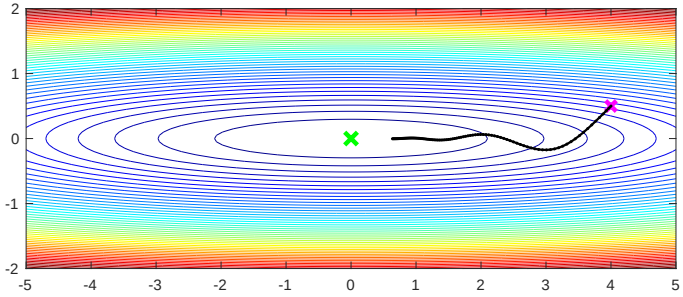
Adam paths for different η_t's



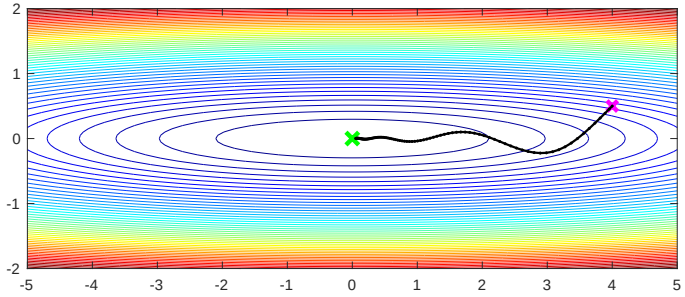
$\eta_t = .001$



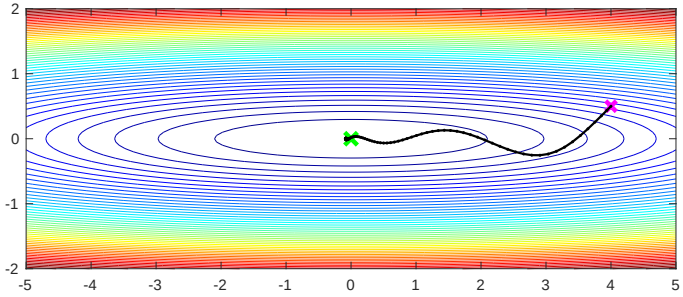
$\eta_t = .005$



$\eta_t = .01$



$\eta_t = .015$



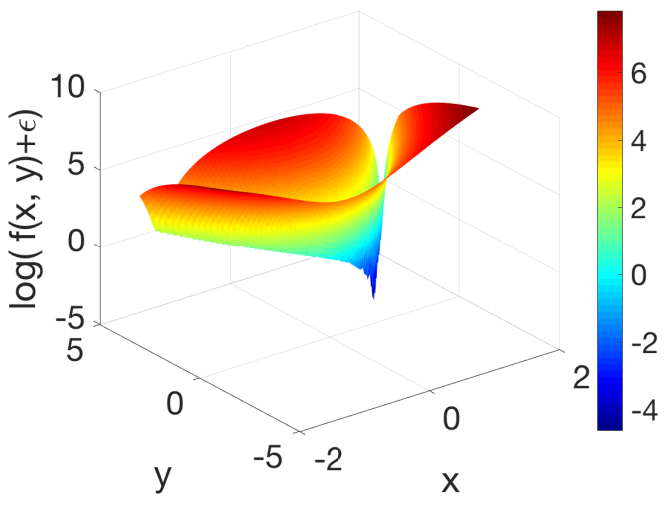
$\eta_t = .02$

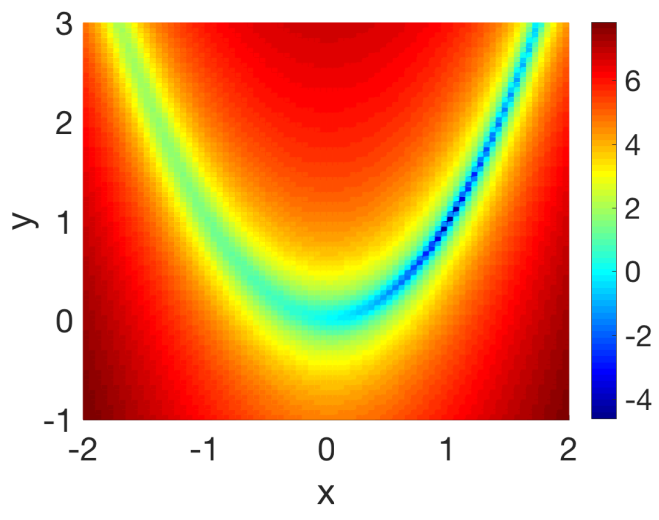
Comparison of performance on the Rosenbrock function

- **2D Rosenbrock function**

$$f(\mathbf{x}) = (x_1 - 1)^2 + 100(x_2 - x_1^2)^2$$

- Visualization of $\log(f(\mathbf{x}) + .01)$

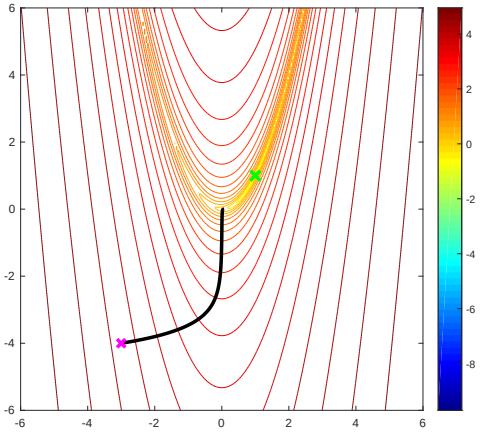




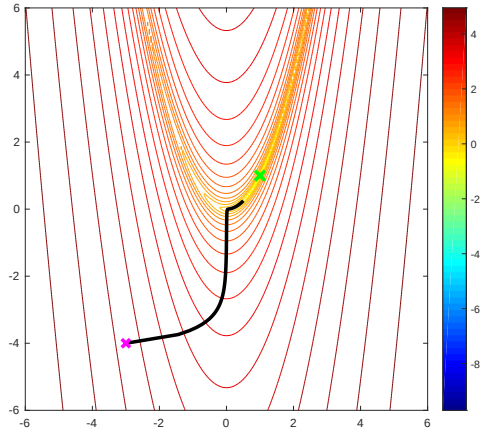
top-down viewpoint

Rosenbrock optimization with GD

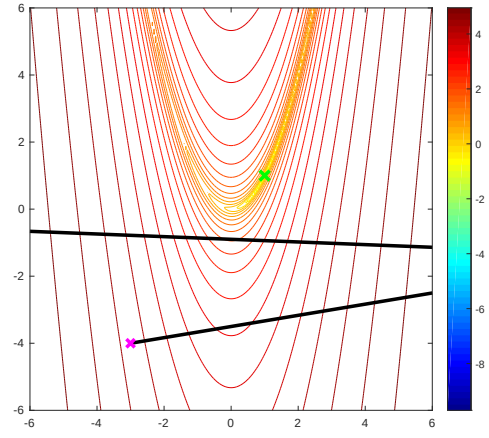
Learning rate: $\eta_0 = .00001$ and # of update steps: $n_{\text{iter}} = 5000$.



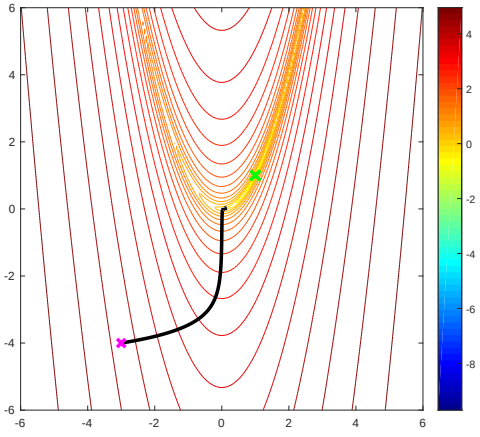
$(\eta_0, n_{\text{iter}})$



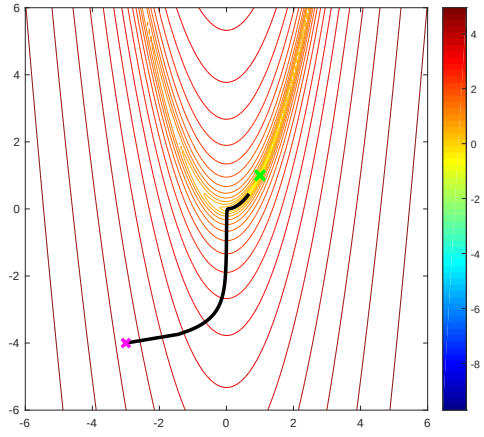
$(10\eta_0, n_{\text{iter}})$



$(100\eta_0, n_{\text{iter}})$



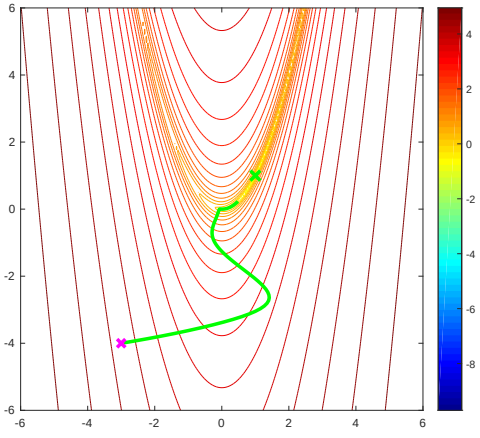
$(\eta_0, 2n_{\text{iter}})$



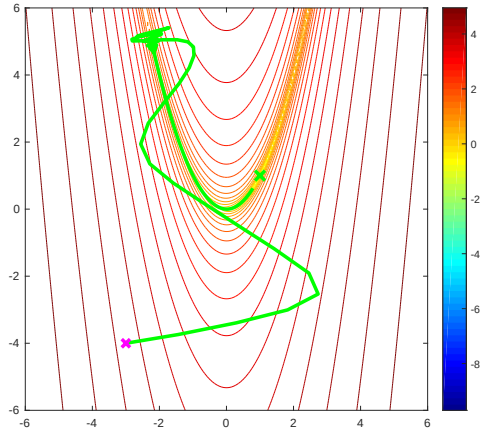
$(10\eta_0, 2n_{\text{iter}})$

Rosenbrock optimization with GD+Momentum

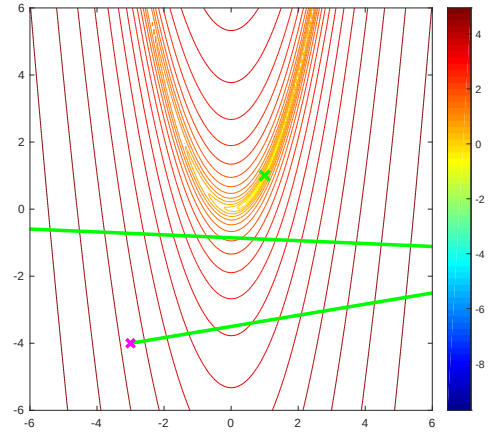
Learning rate: $\eta_0 = .00001$ and # of update steps: $n_{\text{iter}} = 5000$, $\rho = .9$.



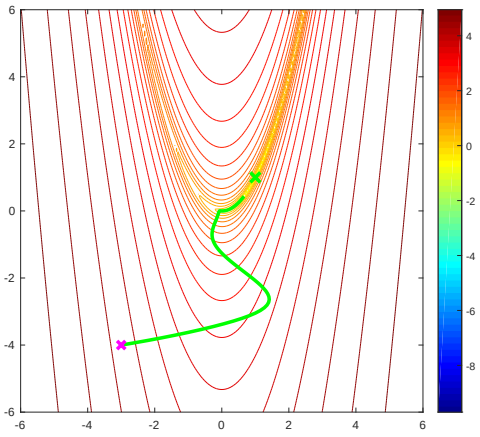
$(\eta_0, n_{\text{iter}})$



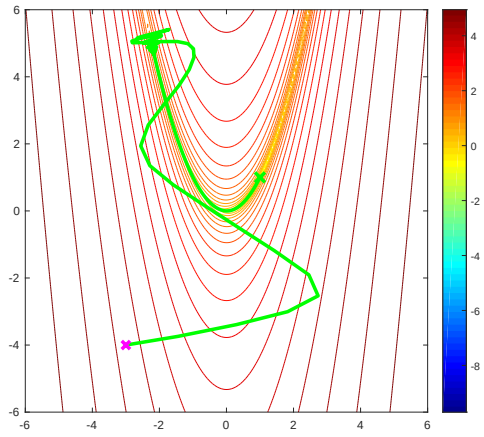
$(10\eta_0, n_{\text{iter}})$



$(100\eta_0, n_{\text{iter}})$



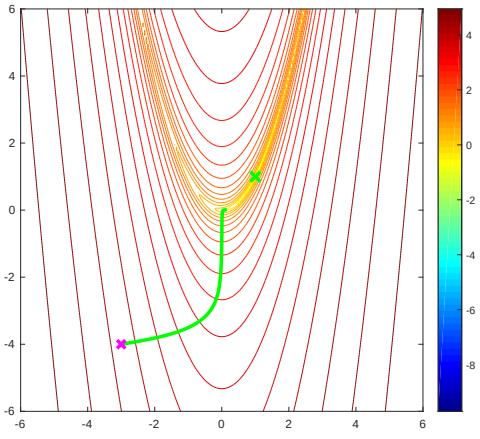
$(\eta_0, 2n_{\text{iter}})$



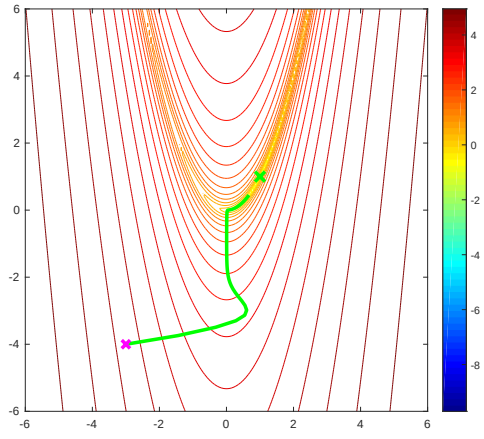
$(10\eta_0, 2n_{\text{iter}})$

Rosenbrock optimization with GD+Momentum

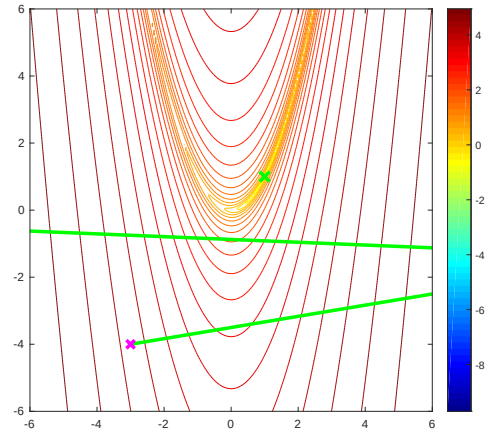
Learning rate: $\eta_0 = .00001$ and # of update steps: $n_{\text{iter}} = 5000$, $\rho = .5$.



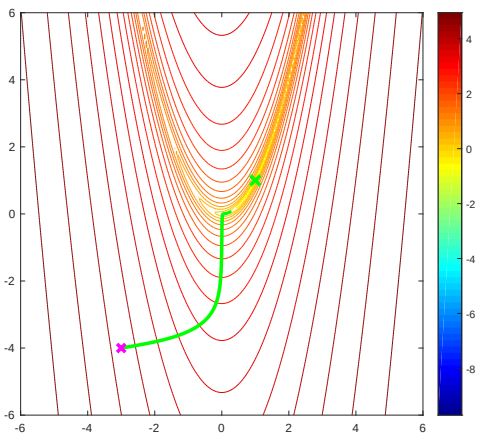
$(\eta_0, n_{\text{iter}})$



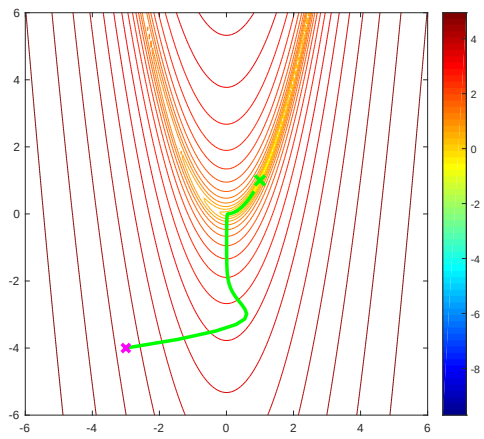
$(10\eta_0, n_{\text{iter}})$



$(100\eta_0, n_{\text{iter}})$



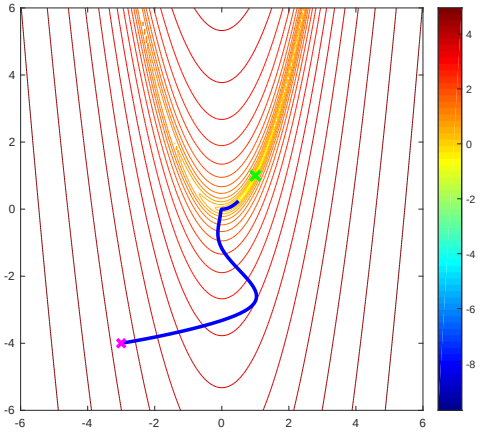
$(\eta_0, 2n_{\text{iter}})$



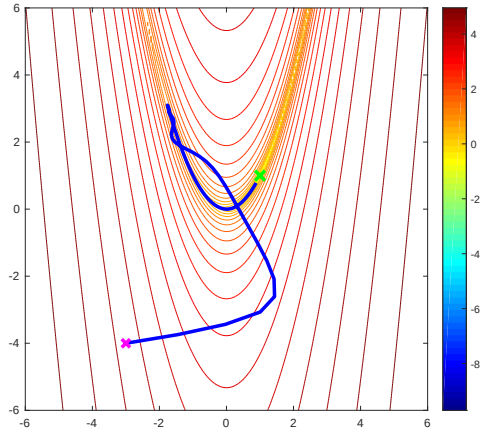
$(10\eta_0, 2n_{\text{iter}})$

Rosenbrock optimization with NAG

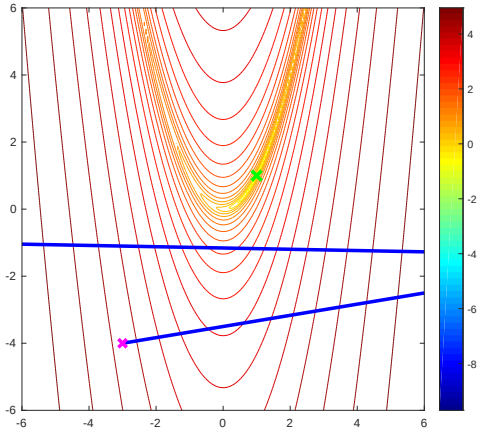
Learning rate: $\eta_0 = .00001$ and # of update steps: $n_{\text{iter}} = 5000$, $\rho = .9$.



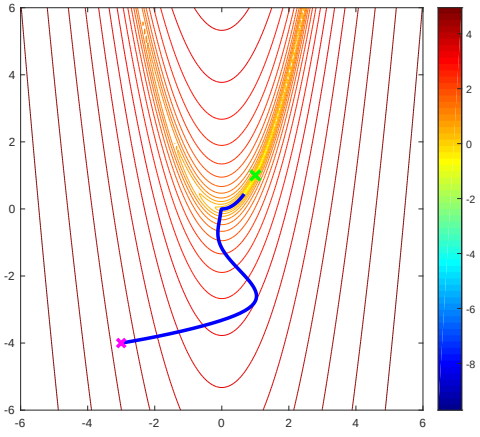
$(\eta_0, n_{\text{iter}})$



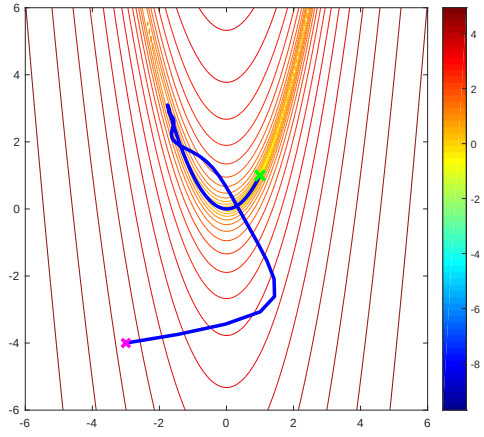
$(10\eta_0, n_{\text{iter}})$



$(100\eta_0, n_{\text{iter}})$



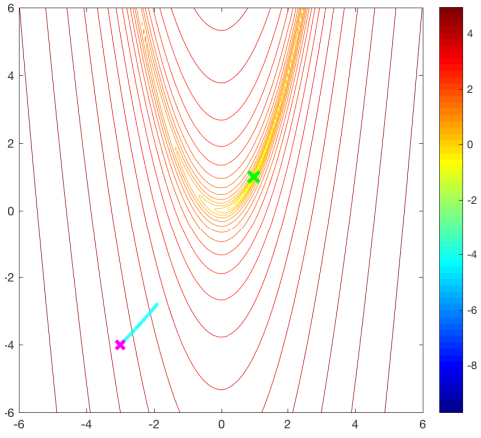
$(\eta_0, 2n_{\text{iter}})$



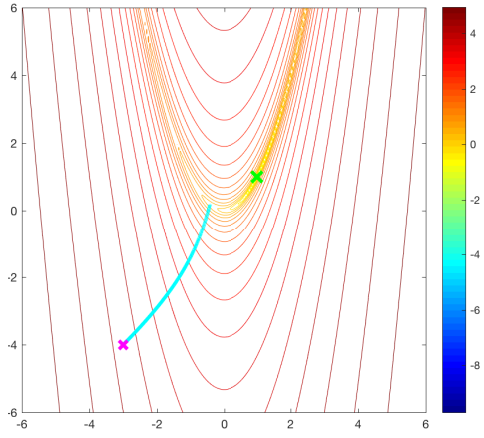
$(10\eta_0, 2n_{\text{iter}})$

Rosenbrock optimization with Adagrad

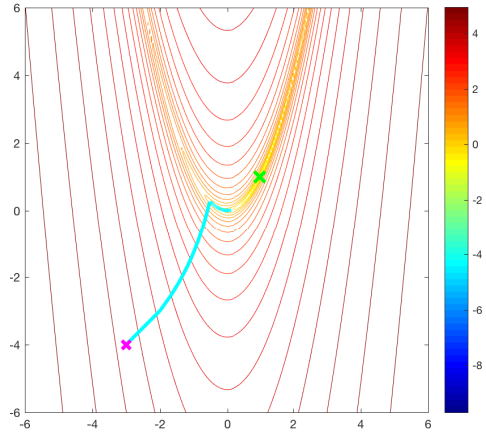
Learning rate: $\eta_0 = .01$ and # of update steps: $n_{\text{iter}} = 5000$.



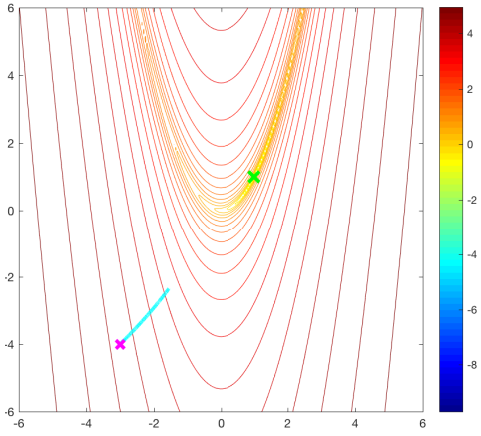
$(\eta_0, n_{\text{iter}})$



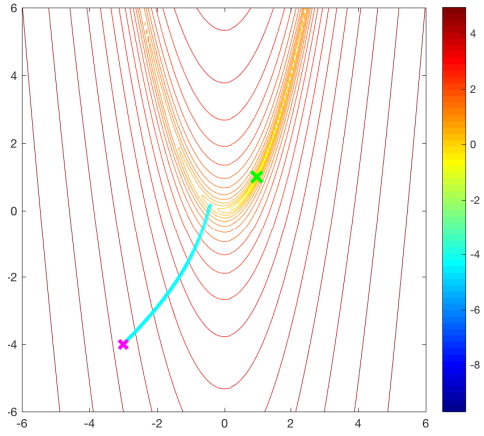
$(10\eta_0, n_{\text{iter}})$



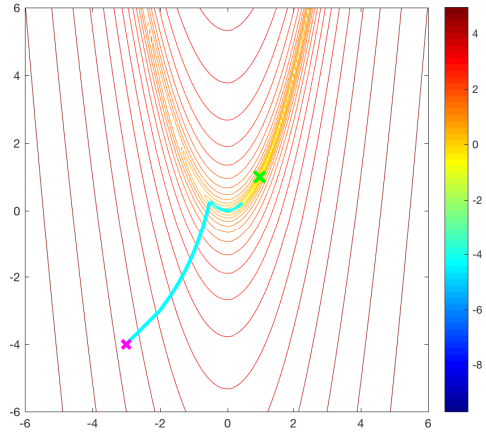
$(100\eta_0, n_{\text{iter}})$



$(\eta_0, 2n_{\text{iter}})$



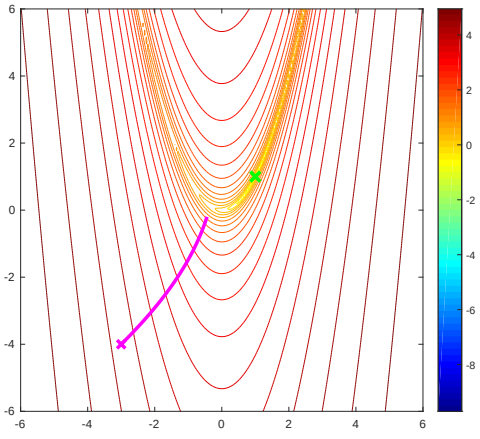
$(10\eta_0, 2n_{\text{iter}})$



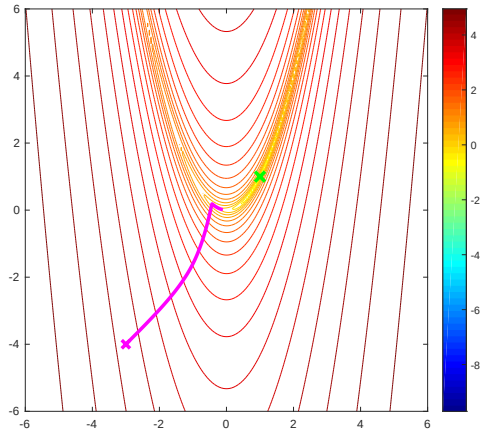
$(100\eta_0, 2n_{\text{iter}})$

Rosenbrock optimization with Adam

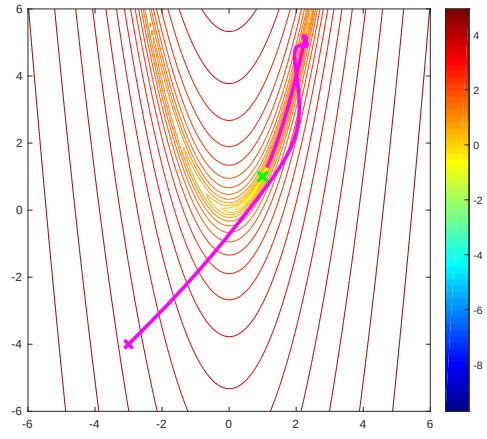
Learning rate: $\eta_0 = .001$ and # of update steps: $n_{\text{iter}} = 5000$.



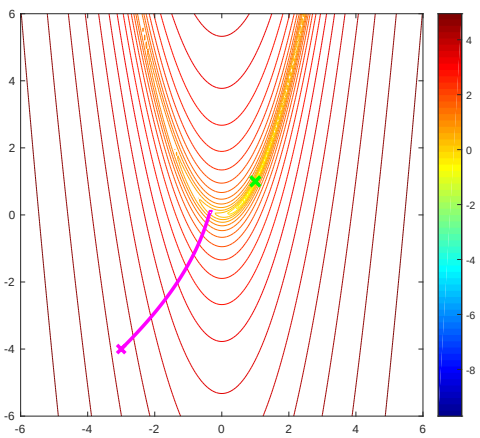
$(\eta_0, n_{\text{iter}})$



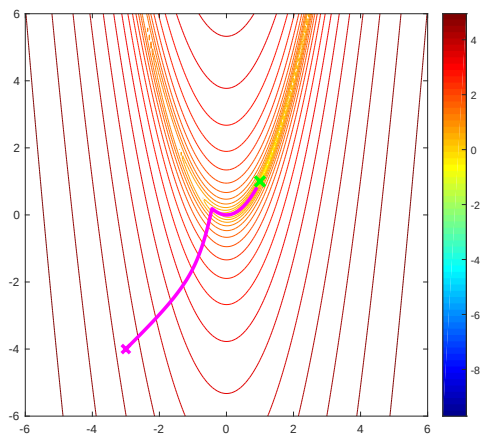
$(10\eta_0, n_{\text{iter}})$



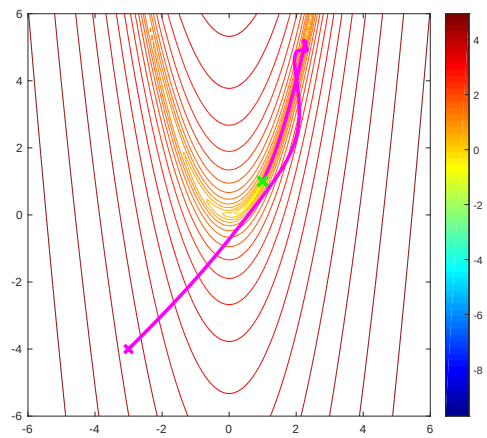
$(100\eta_0, n_{\text{iter}})$



$(\eta_0, 2n_{\text{iter}})$



$(10\eta_0, 2n_{\text{iter}})$

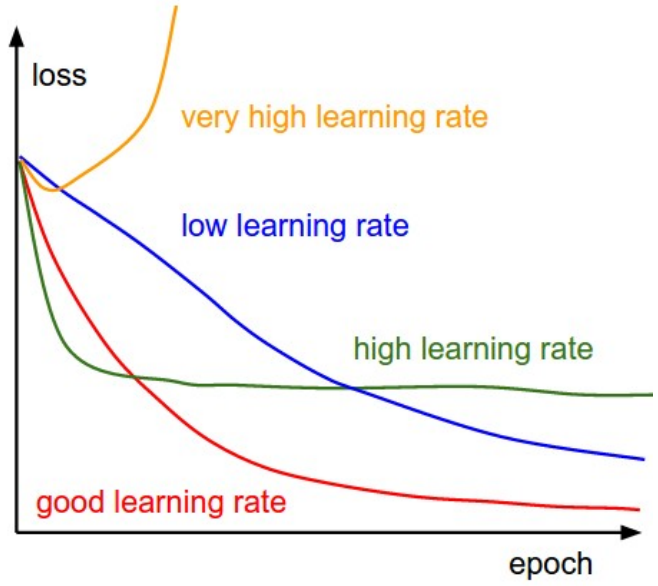


$(100\eta_0, 2n_{\text{iter}})$

Learning Rate Schedules

All the variants have a learning rate

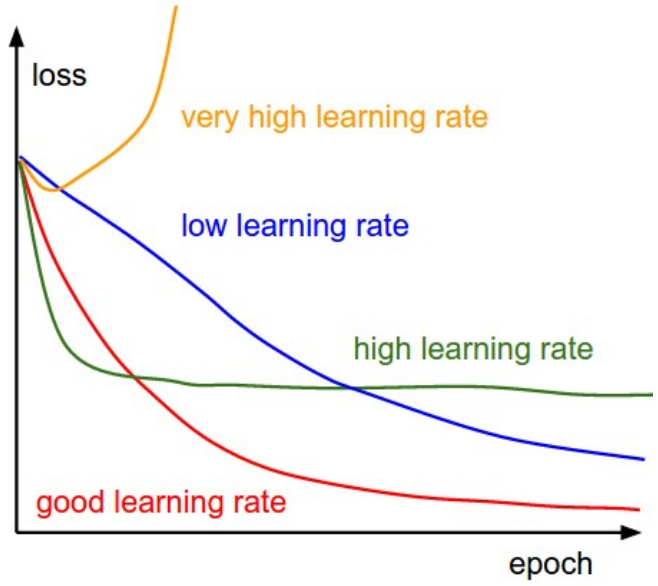
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyper-parameter.



Question: Which one of these learning rates is best to use?

All the variants have a learning rate

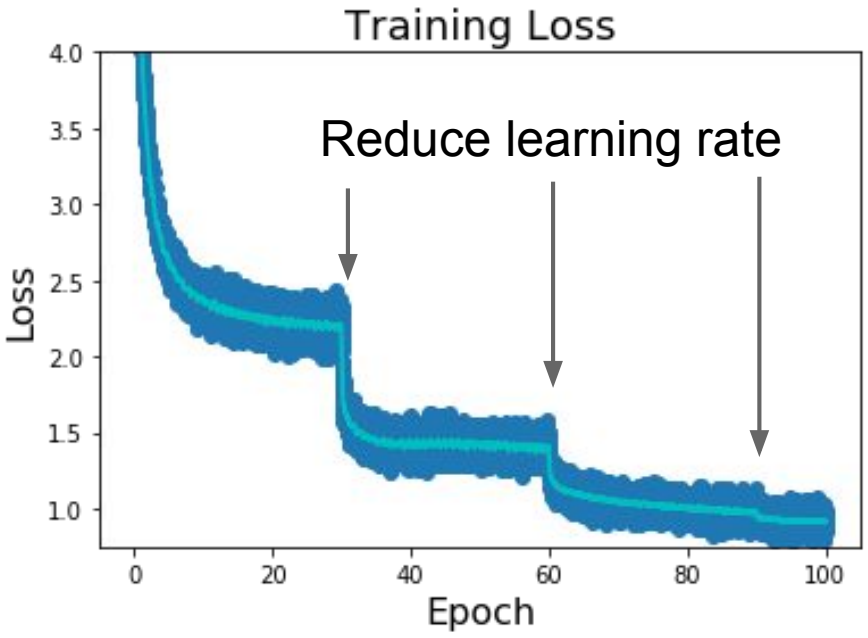
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyper-parameter.



Question: Which one of these learning rates is best to use?

Answer: In reality all of these are good learning rates but depends on stage of training.

Decay learning rate over time

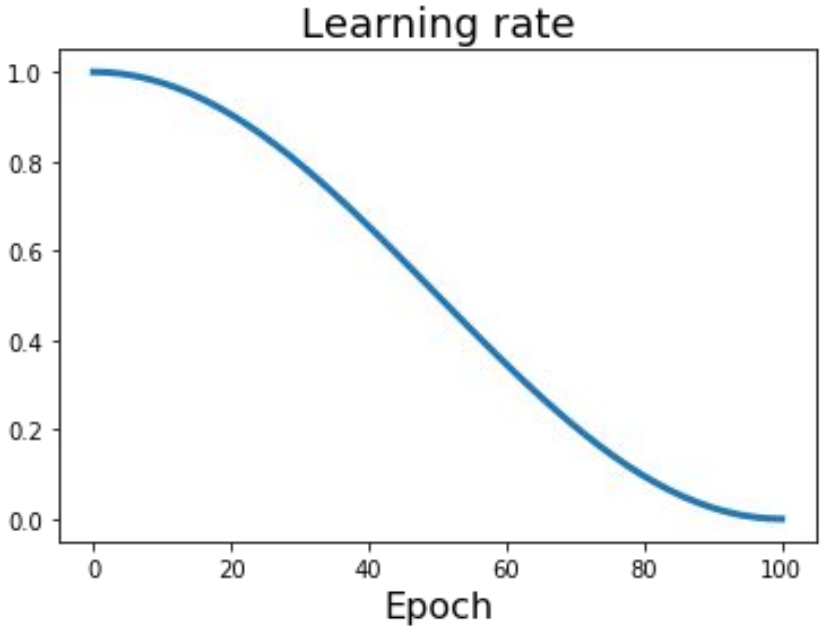


Step Decay:
After every nth epoch set

$$\eta = \alpha\eta$$

where $\alpha \in (0, 1)$. (Instead sometimes people monitor the validation loss and reduce the learning rate when this loss stops improving.)

Decay learning rate over time



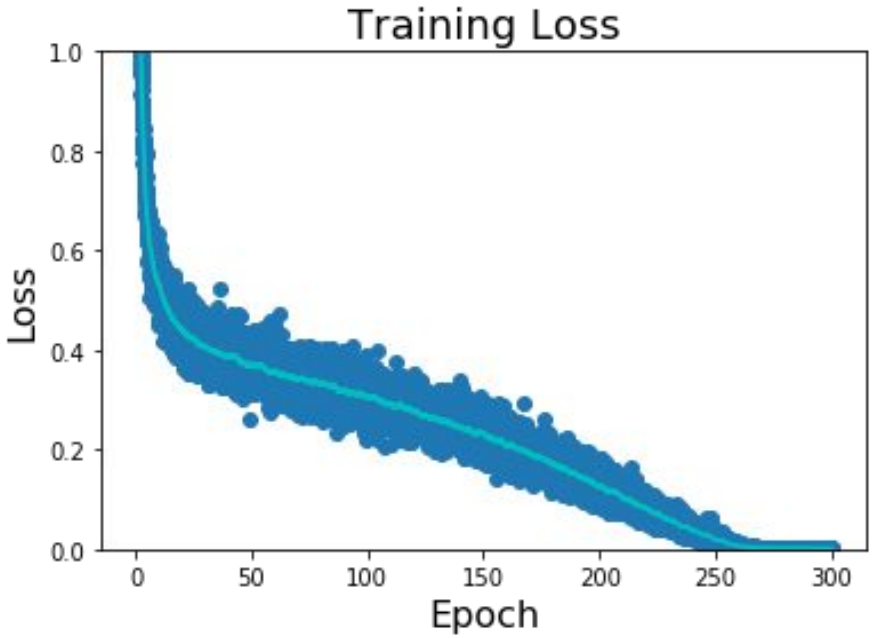
- Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
- Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
- Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
- Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

Cosine Decay:
Learning rate at epoch t given by

$$\eta_t = \frac{1}{2}\eta_0(1 + \cos(t\pi/T))$$

where T is the total number of epochs

Decay learning rate over time



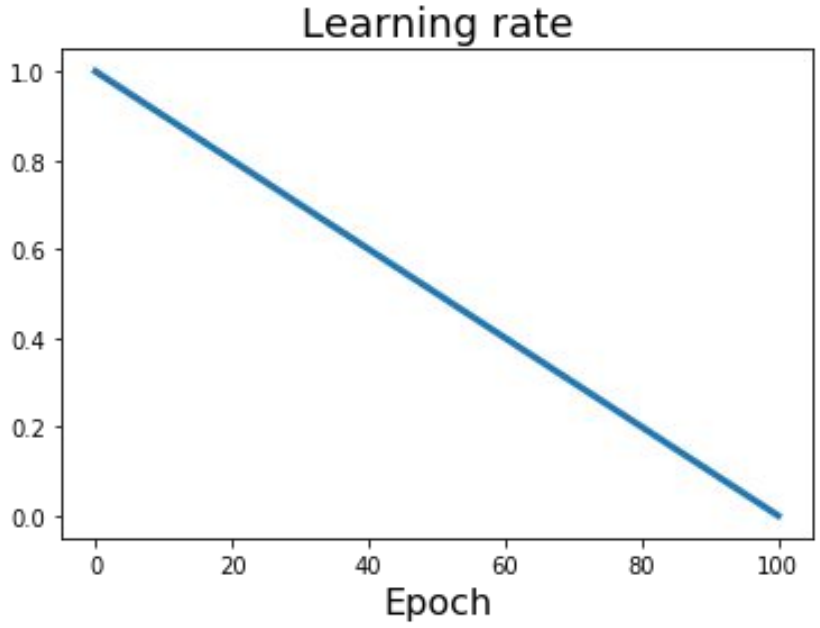
- Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017
- Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018
- Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018
- Child at al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

Cosine Decay:
Learning rate at epoch t given by

$$\eta_t = \frac{1}{2}\eta_0(1 + \cos(t\pi/T))$$

where T is the total number of epochs

Decay learning rate over time

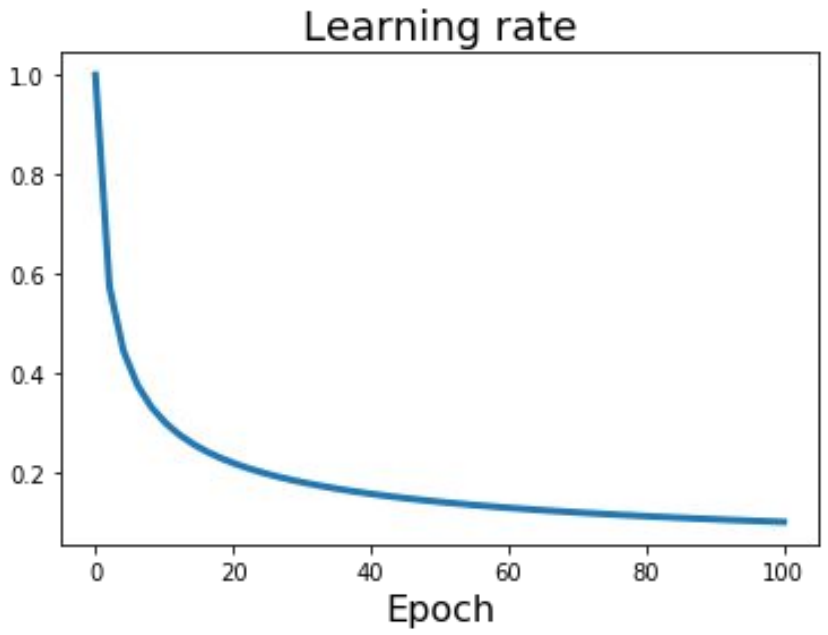


Linear Decay:
Learning rate at epoch t given by

$$\eta_t = \frac{1}{2}\eta_0(1 - t/T)$$

where T is the total number of epochs

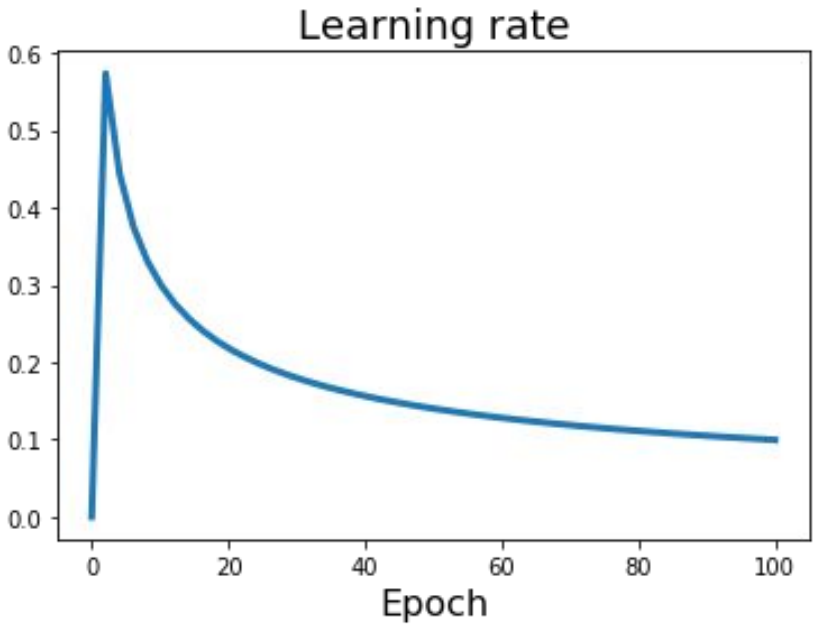
Decay learning rate over time



Inverse square root Decay:
Learning rate at epoch t given by

$$\eta_t = \frac{1}{2}\eta_0/\sqrt{t}$$

Decay learning rate over time: Linear Warmup



- High initial learning rates can make loss explode.
- Linearly increase learning rate from 0 over the first $\sim$5000 iterations to help prevent this.

Size of batch affects learning rate

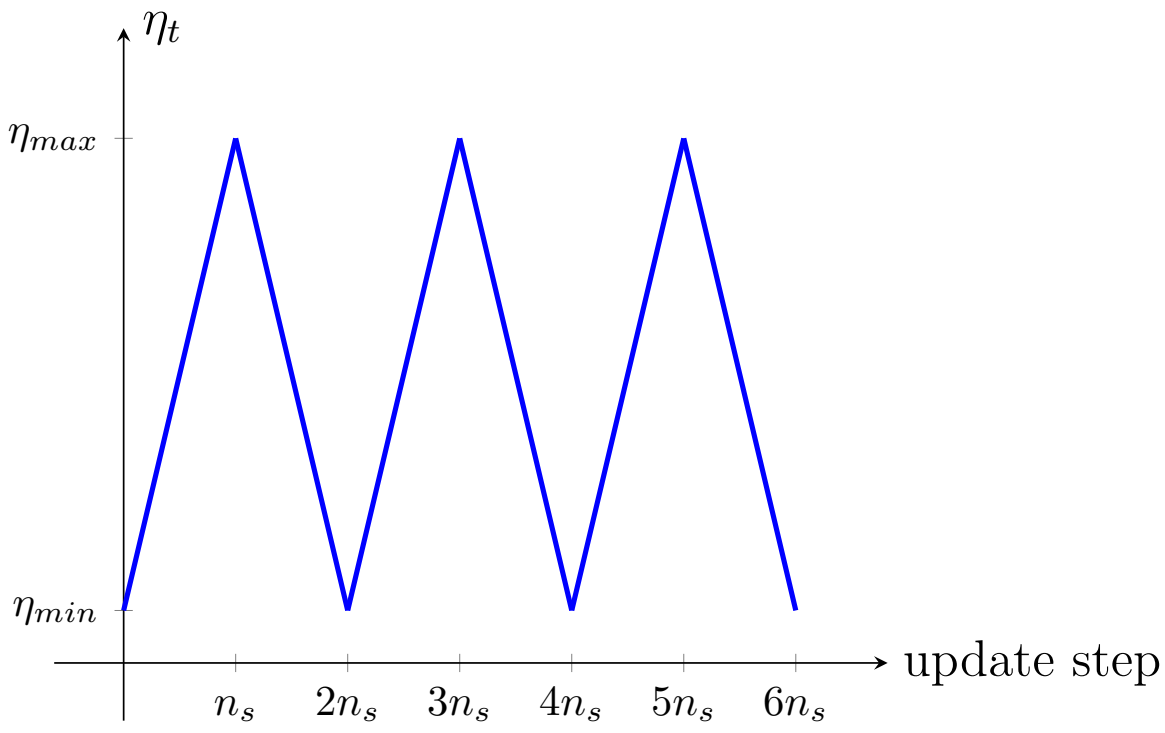
Empirical rule of thumb:
If you increase the batch size by percentage α, also scale the initial learning rate by α

Summary why one wants to anneal the learning rate

- When training deep networks, usually helpful to anneal the learning rate over time.
- **Why?**
  - Stops the parameter vector from bouncing around too widely.
  - $\Longrightarrow$ can reach into deeper, but narrower parts of the loss function.
- But knowing when to decay the learning rate is tricky!

Recent idea: Cyclical learning rates

Cyclical learning rates



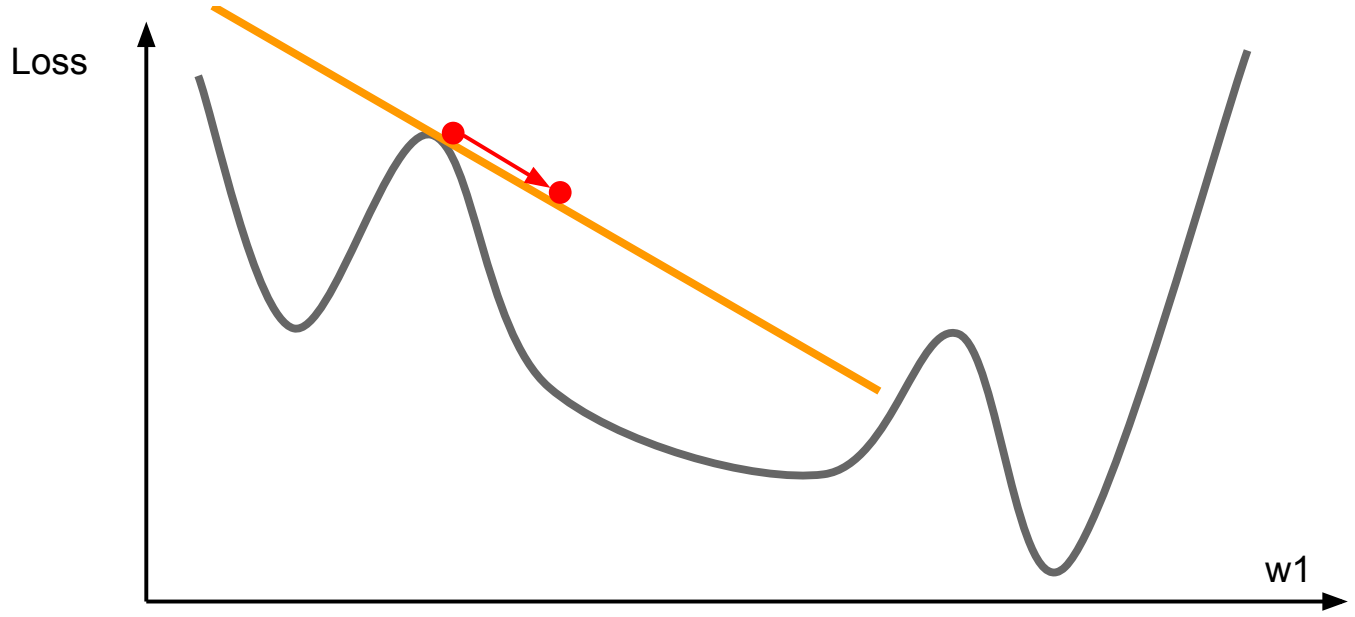
- Vary learning rate between $\eta_{\min}$ and $\eta_{\max}$ and then back to $\eta_{\min}$. Repeat!
- You will explore this approach in Assignment 2.
- Good blog post describing this Setting the learning rate of your neural network.

Which optimizer to use?

- **Adam** is a good default choice in many cases.
  It often works fine/okay even with constant learning rate

- **SGD+Momentum can** outperform **Adam** but usually requires more tuning of learning rate and its schedule
  - Many use warmup + cosine schedule.

Gradient descent (mini-batch gradient descent) is a first-order optimization method



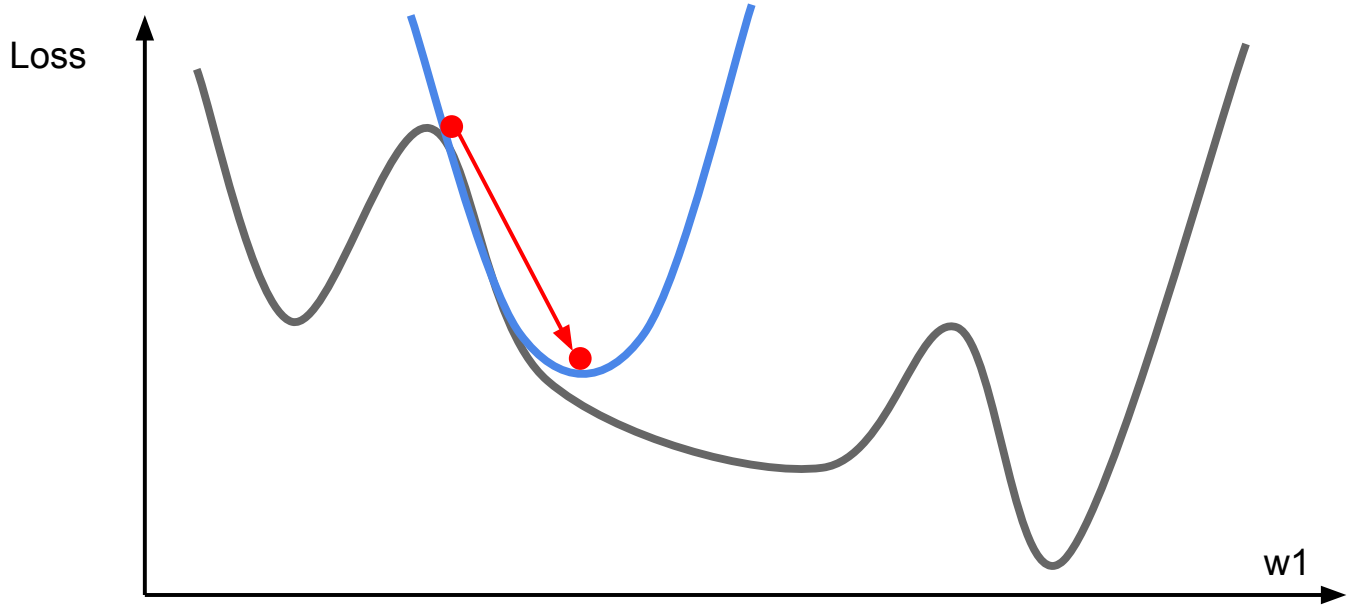
- Approximate the function locally as a linear one using $\nabla_{\mathbf{x}} f(\mathbf{x})$ that is for $\mathbf{x}$ close to $\mathbf{x}_0$ assume

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^T \nabla_{\mathbf{x}} f(\mathbf{x}_0)$$

- Step in direction $-\nabla_{\mathbf{x}} f(\mathbf{x}_0)$ to update estimate of optimal $\mathbf{x}$

Second-order optimisation

More sophisticated optimization algorithms take advantage of second-order information



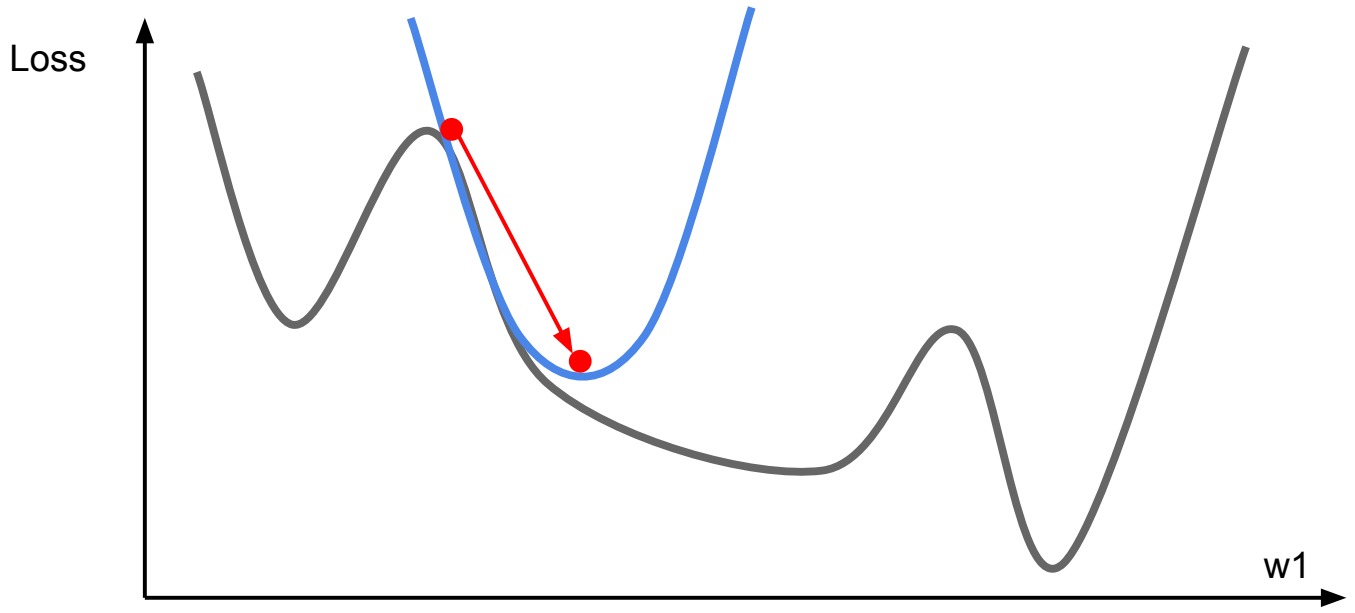
- Approximate the function locally as a quadratic using $\nabla_{\mathbf{x}} f(\mathbf{x})$ and the Hessian that is for $\mathbf{x}$ close to $\mathbf{x}_0$ assume

$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^T \nabla_{\mathbf{x}} f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \nabla^2_{\mathbf{x}} f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

- Step to minima of approximating function: $= \mathbf{x}_0 - \left(\nabla^2_{\mathbf{x}} f(\mathbf{x}_0)\right)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_0)$

Second-order optimisation

Does this approach work for deep learning?



- Approximate the function locally as a quadratic using $\nabla_{\mathbf{x}} f(\mathbf{x})$ and the Hessian that is for $\mathbf{x}$ close to $\mathbf{x}_0$ assume

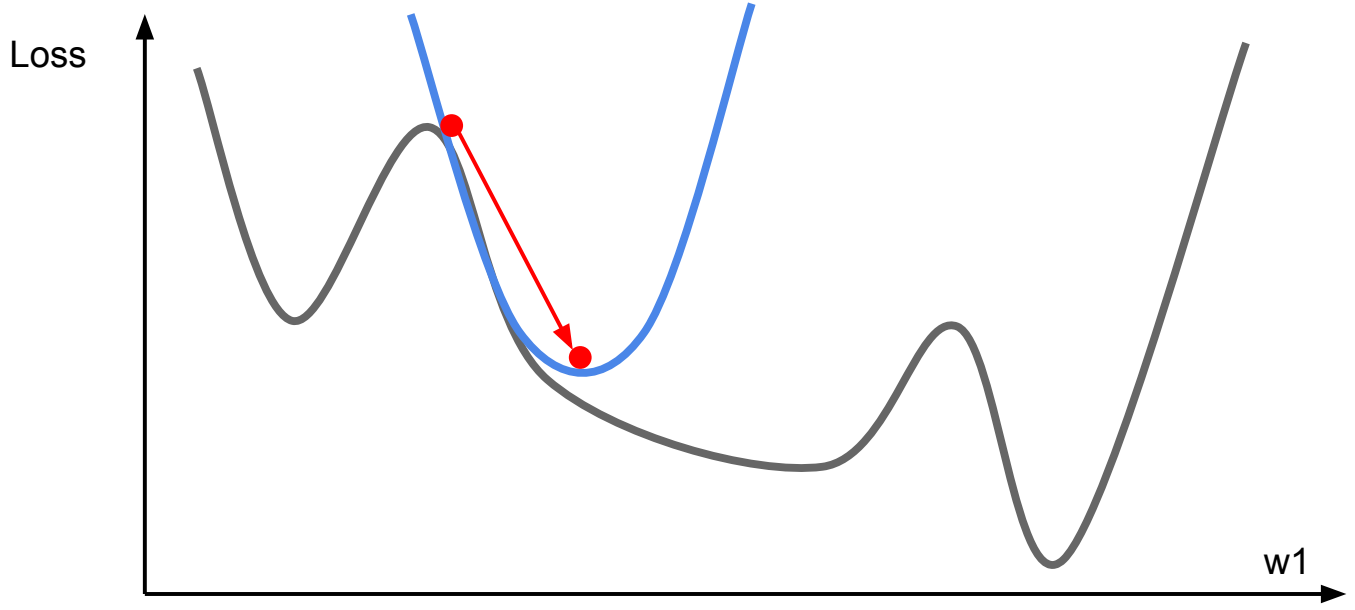
$$f(\mathbf{x}) \approx f(\mathbf{x}_0) + (\mathbf{x} - \mathbf{x}_0)^T \nabla_{\mathbf{x}} f(\mathbf{x}_0) + \frac{1}{2}(\mathbf{x} - \mathbf{x}_0)^T \nabla^2_{\mathbf{x}} f(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0)$$

- Step to minima of approximating function: $= \mathbf{x}_0 - \left(\nabla^2_{\mathbf{x}} f(\mathbf{x}_0)\right)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_0)$

Second-order optimisation

Does this approach work for deep learning? **Not feasible**

- Hessian has $O(p^2)$ elements.
- Inverting takes $O(p^3)$
- p typically much greater than 10^6



- Step to minima of approximating function: $= \mathbf{x}_0 - \left(\nabla^2_{\mathbf{x}} f(\mathbf{x}_0)\right)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_0)$

Does this approach work for deep learning?

Not feasible to invert the Hessian

$$\mathbf{x}_1 = \mathbf{x}_0 - \left(\nabla^2_{\mathbf{x}} f(\mathbf{x}_0)\right)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_0)$$

But approximate second-order optimization algorithms exist

- **Quasi-Newton methods** (BGFS most popular):
  Instead of inverting the Hessian ($O(p^3)$), approximate inverse Hessian with rank 1 updates over time ($O(p^2)$ each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

- Unfortunately as of yet **L-BFGS** does not transfer very well to mini-batch setting. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Does this approach work for deep learning?

Not feasible to invert the Hessian

$$\mathbf{x}_1 = \mathbf{x}_0 - \left(\nabla^2_{\mathbf{x}} f(\mathbf{x}_0)\right)^{-1} \nabla_{\mathbf{x}} f(\mathbf{x}_0)$$

But approximate second-order optimization algorithms exist

- **Quasi-Newton methods** (BGFS most popular):
  Instead of inverting the Hessian ($O(p^3)$), approximate inverse Hessian with rank 1 updates over time ($O(p^2)$ each).

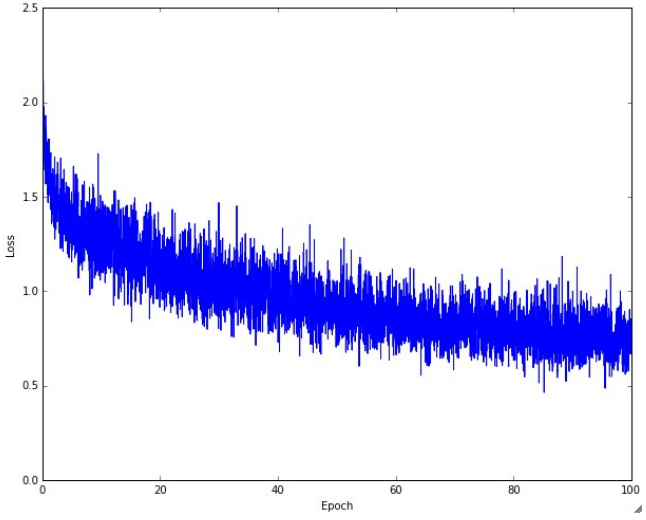
- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

- Unfortunately as of yet **L-BFGS** does not transfer very well to mini-batch setting. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

Improve Test Error

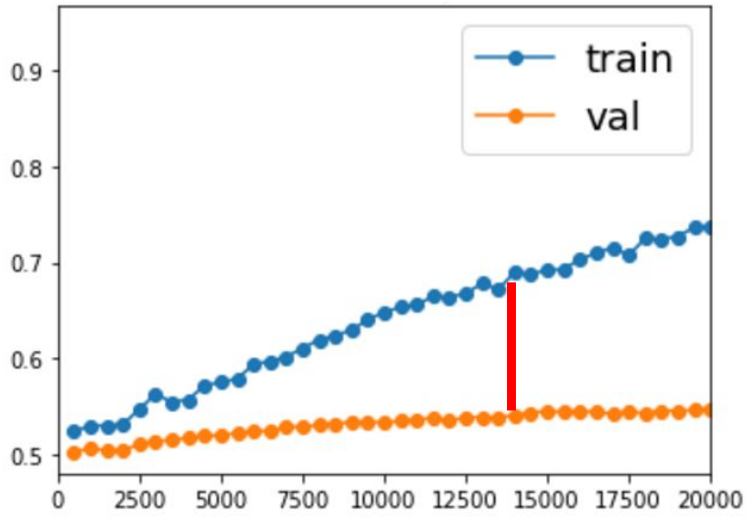
Beyond Training Error

Train Loss



Better optimization algorithms help reduce training loss.

Train and Val Acc



But really care about error on new data - how to reduce the gap?

Regularization: Add regularizer to loss

$$J(\mathcal{D}, \Theta) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x},y)\in\mathcal{D}} l(f(\mathbf{x}, \Theta), y) + \lambda R(\Theta)$$

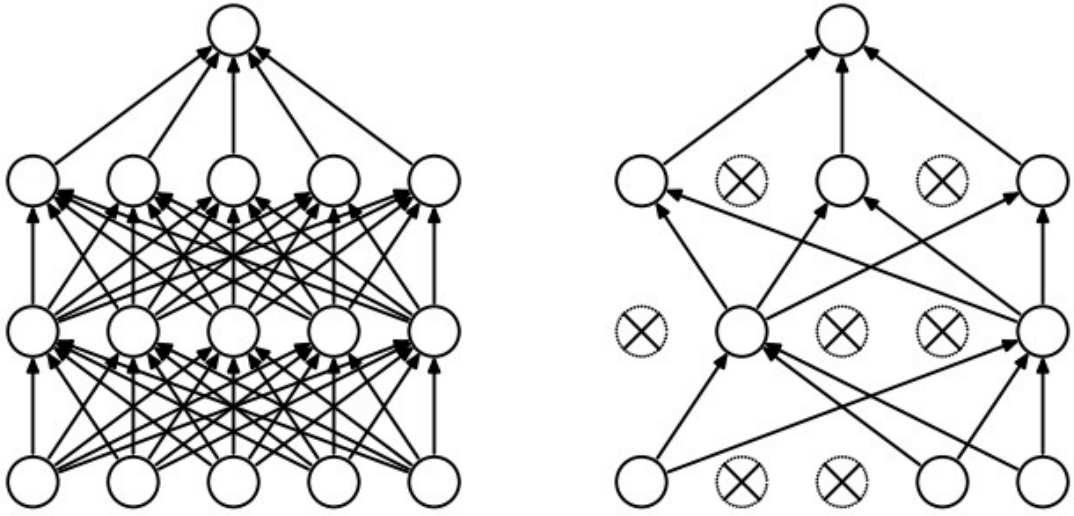
In common use

- L2 regularization $R(W) = \sum_k \sum_l W_{k,l}^2$
- L1 regularization $R(W) = \sum_k \sum_l |W_{k,l}|$
- Elastic Net (L1 + L2) $R(W) = \sum_k \sum_l \left(\beta W_{k,l}^2 + |W_{k,l}|\right)$

Regularization Via **Dropout**

Dropout

- **Randomly set some activations to zero in forward pass**



(a) Standard Neural Net (b) After applying dropout.

[Srivastava et al.]

- Probability of dropping an activation is a hyper-parameter.
- Training practice introduced by Hinton.

  Srivastava et al, "Dropout: A simple way to prevent neural networks from overfitting", JMLR 2014

- **Note**: Each training sample in the mini-batch has its own random dropout mask.

Training with Dropout

Set $p \in (0, 1]$ ← probability of keeping an activation active.

The Forward Pass (for a 3-layer network)

1. Compute the first set of activation values:

$$\mathbf{x}^{(1)} = \max(0, W_1 \mathbf{x}^{(0)} + \mathbf{b}_1)$$

2. Randomly choose which entries of $\mathbf{x}^{(1)}$ to drop

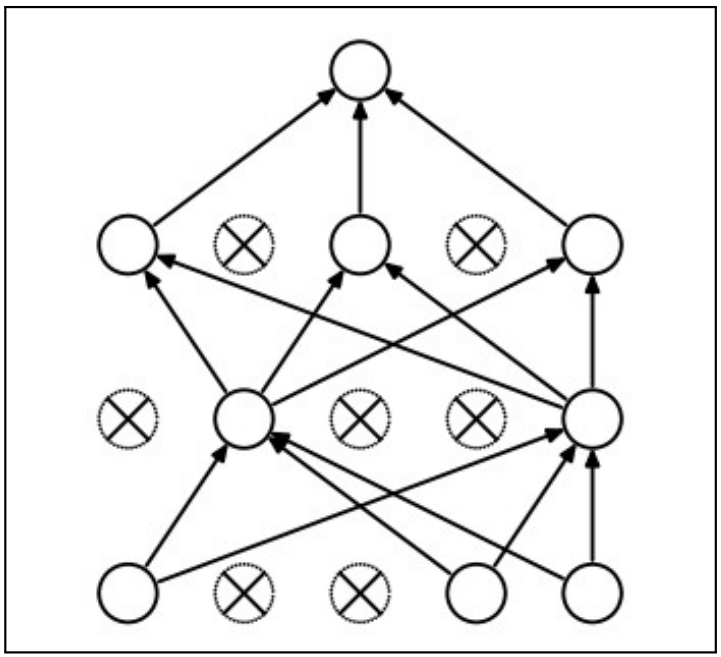
$$\mathbf{u}_1 = \Big(\text{rand}(\text{size}(\mathbf{x}^{(1)})) < p\Big) \quad \text{(Matlab notation)}$$
$$\mathbf{x}^{(1)} = \mathbf{x}^{(1)} .* \mathbf{u}_1$$

3. Repeat the process for the next layer

$$\mathbf{x}^{(2)} = \max(0, W_2 \mathbf{x}^{(1)} + \mathbf{b}_2)$$
$$\mathbf{u}_2 = \Big(\text{rand}(\text{size}(\mathbf{x}^{(2)})) < p\Big) \quad \text{(Matlab notation)}$$
$$\mathbf{x}^{(2)} = \mathbf{x}^{(2)} .* \mathbf{u}_2$$

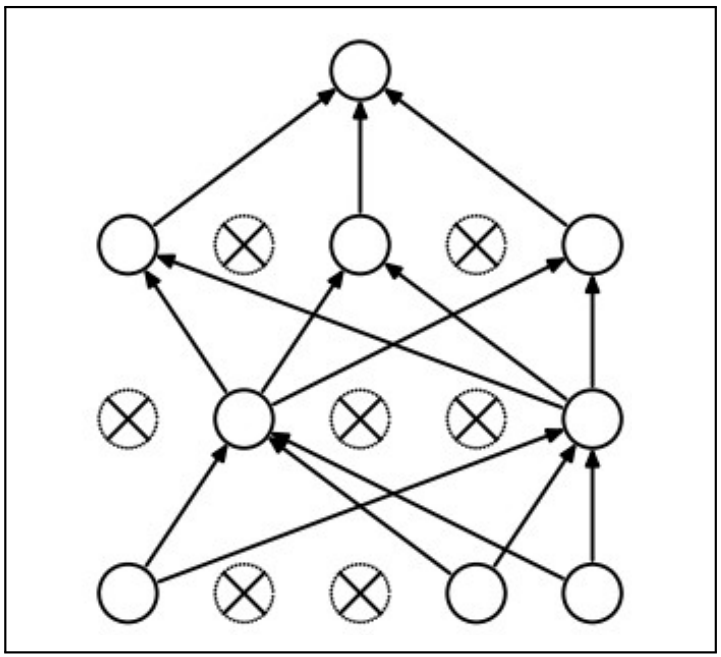
4. Output: $\mathbf{x}^{(3)} = \text{SoftMax}(W_3 \mathbf{x}^{(2)} + \mathbf{b}_3)$

Why is this a good idea?



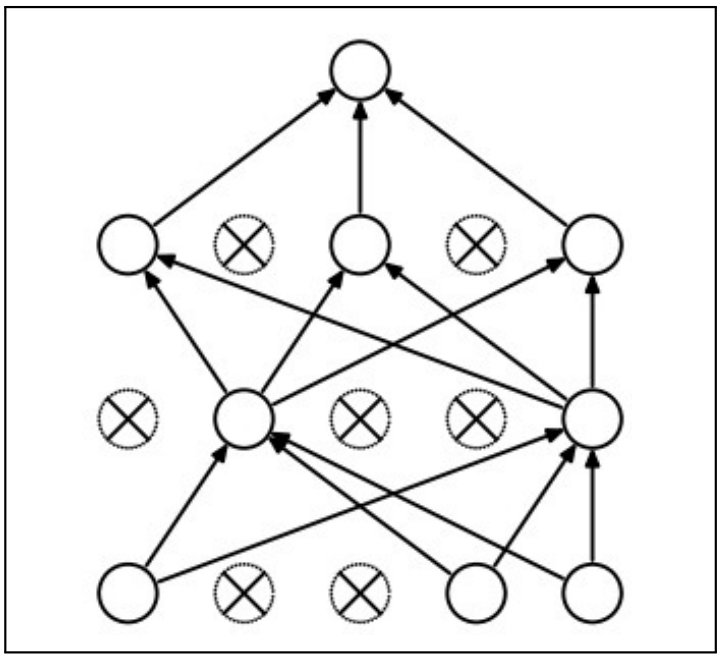
- Forces the network to have a redundant representation.
- Prevents co-adaptation of features.
- **Another interpretation**
  - Dropout is training a large ensemble of models.
  - Each binary mask is one model, gets trained on only ∼one datapoint.

Why is this a good idea?



- Forces the network to have a redundant representation.
- Prevents co-adaptation of features.
- **Another interpretation**
  - Dropout is training a large ensemble of models.
  - Each binary mask is one model, gets trained on only $\sim$one datapoint.

At test time

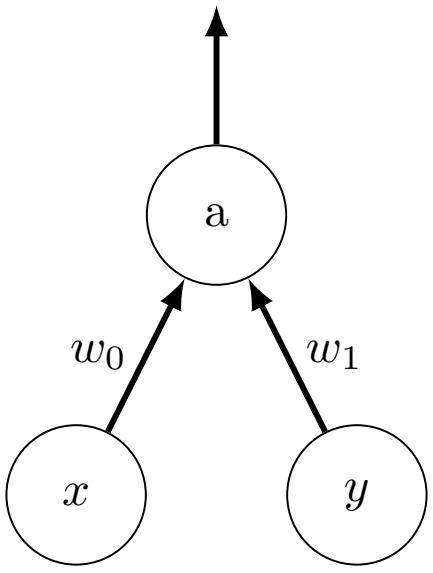


- Dropout introduces randomness into the output.
- Should integrate over all possible masks at test-time but...
- **Monte Carlo approximation**
  - Do many forward passes with different dropout masks.
  - Average all the predictions.

At test time

- Can do this with a single forward pass (approximately).
- Leave all the activations turned on (no dropout).
- Surely we must compensate?

At test time

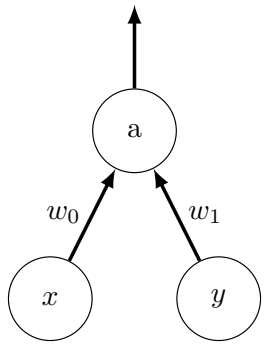


Consider this simple partial network

- During testing if we do not compensate:

$$a_{\text{test}} = w_0 x + w_1 y$$

At test time



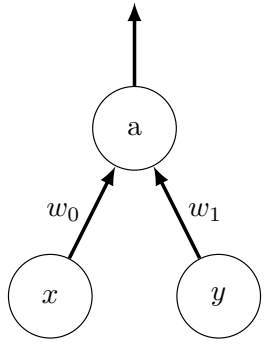
Consider this simple partial network

- During dropout training:
  1. Each input activation x and y is switched off with probability $1 - p$.
  2. The possible input activations are

inputs	probability
$(0, 0)$	$(1-p)^2$
$(x, 0)$	$p(1-p)$
$(0, y)$	$(1-p)p$
(x, y)	p^2

  3. $$\begin{aligned} \mathsf{E}[a_{\text{training}}] &= (1-p)^2(w_0 0 + w_1 0) + p(1-p)(w_0 x + w_1 0) \\ &\quad + p(1-p)(w_0 0 + w_1 y) + p^2(w_0 x + w_1 y) \\ &= p\,(w_0 x + w_1 y) \\ &= p\,a_{\text{test}} \end{aligned}$$

At test time



Consider this simple partial network

- During **testing** if we do not compensate:

$$a_{\text{test}} = w_0 x + w_1 y$$

- During **dropout training**:

$$\mathsf{E}[a_{\text{training}}] = p\,(w_0 x + w_1 y) = p\,a_{\text{test}}$$

$\Longrightarrow$ have to compensate at test time by scaling the activations by p.

Must compensate at test time

- Must scale the activations so for each neuron:
  output at test time = expected output at training time

- Don't drop activations but have to compensate

$$\mathbf{x}^{(1)} = \max(0, W_1\mathbf{x}^{(0)} + \mathbf{b}_1) * p$$
$$\mathbf{x}^{(2)} = \max(0, W_2\mathbf{x}^{(1)} + \mathbf{b}_2) * p$$
$$\mathbf{x}^{(3)} = \mathrm{SoftMax}(W_3\mathbf{x}^{(2)} + \mathbf{b}_3)$$

More common: Inverted Dropout

- During **training**:

$$\mathbf{x}^{(1)} = \max(0, W_1\mathbf{x}^{(0)} + \mathbf{b}_1)$$
$$\mathbf{u}_2 = (\text{rand}(\text{size}(\mathbf{x}^{(1)})) < p)/p \quad \longleftarrow \text{Note } /p$$
$$\mathbf{x}^{(1)} = \mathbf{x}^{(1)} .* \mathbf{u}_2$$

$$\mathbf{x}^{(2)} = \max(0, W_2\mathbf{x}^{(1)} + \mathbf{b}_2)$$
$$\mathbf{u}_2 = (\text{rand}(\text{size}(\mathbf{x}^{(2)})) < p)/p \quad \longleftarrow \text{Note } /p$$
$$\mathbf{x}^{(2)} = \mathbf{x}^{(2)} .* \mathbf{u}_3$$

$$\mathbf{x}^{(3)} = \text{SoftMax}(W_3\mathbf{x}^{(2)} + \mathbf{b}_3)$$

- $\Longrightarrow$ At **test time** no scaling necessary:

$$\mathbf{x}^{(1)} = \max(0, W_1\mathbf{x}^{(0)} + \mathbf{b}_1)$$
$$\mathbf{x}^{(2)} = \max(0, W_2\mathbf{x}^{(1)} + \mathbf{b}_2)$$
$$\mathbf{x}^{(3)} = \text{SoftMax}(W_3\mathbf{x}^{(2)} + \mathbf{b}_3)$$

Many more regularization strategies exist

Will introduce more of the popular ones in the lectures next week.

- Data augmentation
- Stochastic depth
- Mix-up
- Cutout
- .....

Evaluation: Model Ensembles

Model Ensembles

- Train multiple independent models (same hyper-parameter settings, different initializations). ($\sim$ 5 models)
- At test time apply each model and average their results.

Model Ensemble on the cheap

- Can also get a small boost from averaging multiple model checkpoints of a single model.
- At test time apply each model and average their results.

Choosing Hyper-parameters
(without tons of GPUs)

Training neural networks not completely trivial!

- Several **hyper-parameters** affect the quality of your training.
- These include
  - learning rate
  - degree of regularization
  - network architecture
  - hyper-parameters controlling weight initialization
- If these (potentially correlated) hyper-parameters are not appropriately set $\Longrightarrow$ you will not learn an **effective** network.
- Multiple quantities you should monitor during training.
- These quantities indicate
  - a reasonable hyper-parameter setting and/or
  - how hyper-parameters setting could be changed for the better.

Choosing hyper-parameters

1. Check initial loss

   At initialization for multi-class classification with SoftMax and C classes the loss should be approx $\log(C)$.

2. Overfit a small sample set

   Try to train to 100% training accuracy on a small sample of training data ($\sim$5-10 mini-batches) - play around with architecture, learning rate, weight initialization

   Loss not going down? LR too low and/or bad initialization

   Loss explodes to Inf or NaN? LR too high and/or bad initialization.

Choosing hyper-parameters

1. Check initial loss

   At initialization for multi-class classification with SoftMax and C classes the loss should be approx $\log(C)$.

2. Overfit a small sample set

   Try to train to 100% training accuracy on a small sample of training data ($\sim$5-10 mini-batches) - play around with architecture, learning rate, weight initialization

3. Find LR that makes loss go down

   Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within $\sim$100 iterations

   Good learning rates to try: 1e-1, 1e-2, 1e-3, 1e-4

Choosing hyper-parameters

1. Check initial loss

   At initialization for multi-class classification with SoftMax and C classes the loss should be approx $\log(C)$.

2. Overfit a small sample set

   Try to train to 100% training accuracy on a small sample of training data ($\sim$5-10 mini-batches) - play around with architecture, learning rate, weight initialization

3. Find LR that makes loss go down

   Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within $\sim$100 iterations

4. Coarse grid, train for $\sim$1-5 epochs

   Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for $\sim$1-5 epochs.

   Good weight decay to try: 1e-4, 1e-5, 0

Choosing hyper-parameters

1. Check initial loss
2. Overfit a small sample set

   Try to train to 100% training accuracy on a small sample of training data ($\sim$5-10 mini-batches) - play around with architecture, learning rate, weight initialization
3. Find LR that makes loss go down

   Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within $\sim$100 iterations
4. Coarse grid search, train for $\sim$1-5 epochs

   Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for $\sim$1-5 epochs.
5. Refine grid search, train longer

   Pick best models from Step 4, train them for longer ($\sim$10-20 epochs).

Choosing hyper-parameters

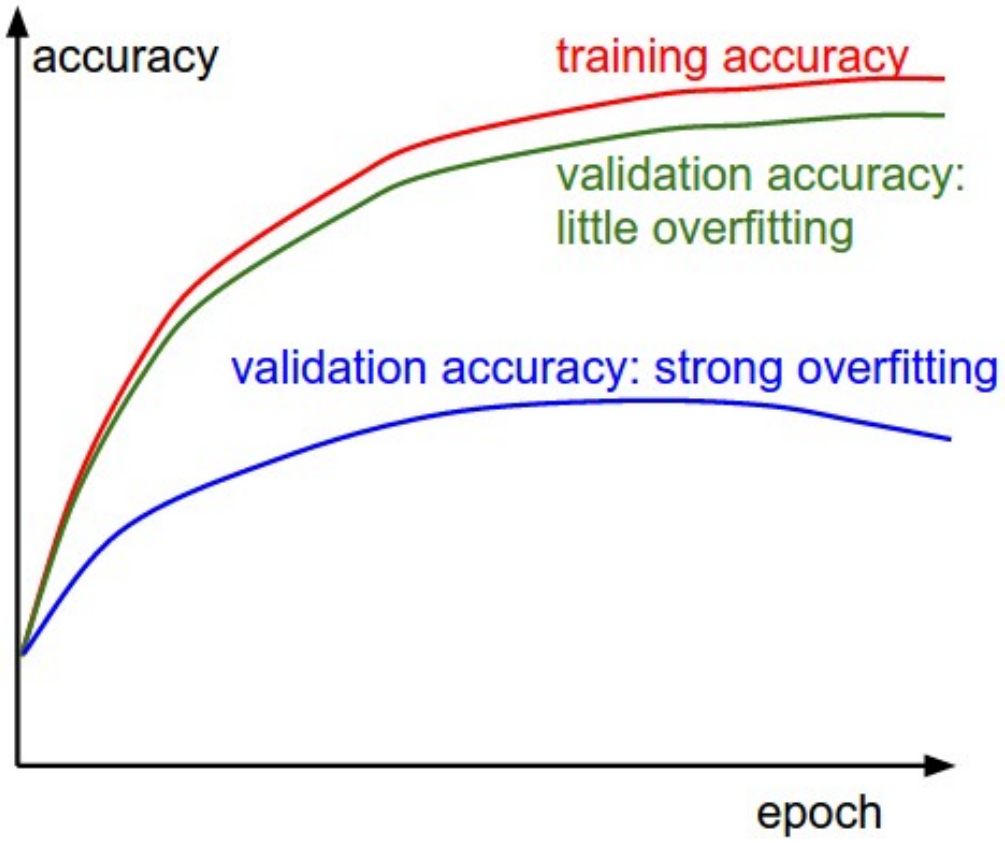
1. Check initial loss
2. Overfit a small sample set
3. Find LR that makes loss go down

   Use the architecture from the previous step, use all training data, turn on small weight decay, find a learning rate that makes the loss drop significantly within ~100 iterations
4. Coarse grid search, train for ~1-5 epochs

   Choose a few values of learning rate and weight decay around what worked from Step 3, train a few models for ~1-5 epochs.
5. Refine grid search, train longer

   Pick best models from Step 4, train them for longer (~10-20 epochs).
6. Look at loss and accuracy curves

Monitor & visualize the accuracy

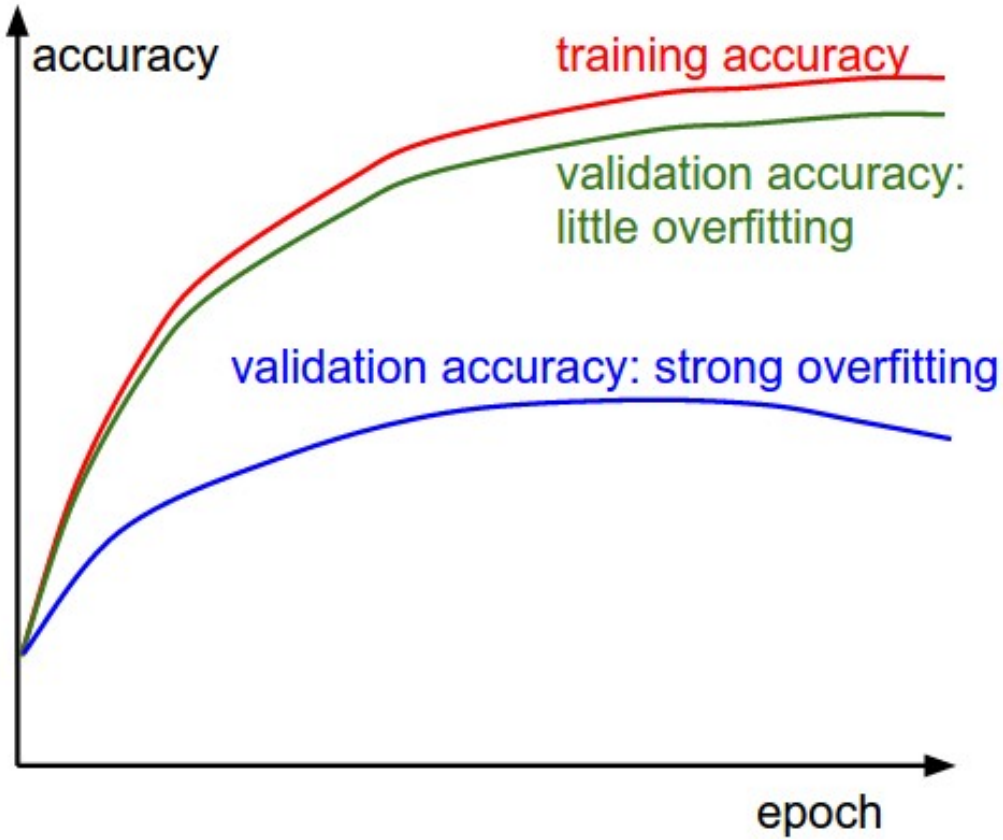


Gap between training and validation accuracy indicates amount of over-fitting.

Under-fitting $\Longrightarrow$ model capacity not high enough:

- Increase the size of the network.

Monitor & visualize the accuracy



Gap between training and validation accuracy indicates amount of over-fitting.

Over-fitting $\Longrightarrow$ should increase regularization during training:

- Increase the degree of L_2 regularization.
- More dropout.
- Use more training data.

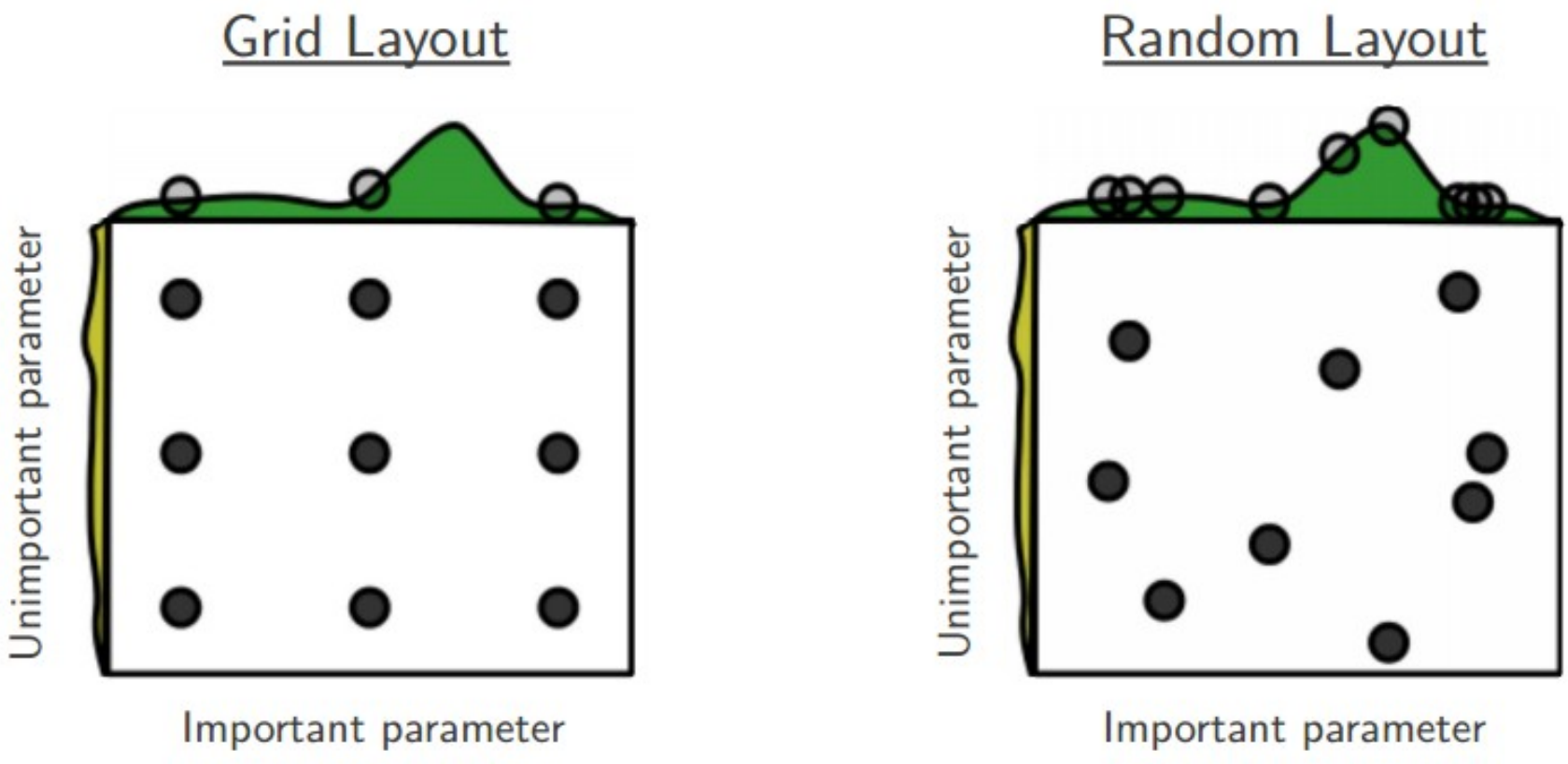
Further note on hyper-parameter ranges

- Search for the **learning-rate** and **regularization** hyperparameters on a log scale.

  Example:
  Generate a potential learning rate with

$$\alpha = \text{uniform}(-6, 1)$$
$$\eta = 10^{\alpha}$$

Prefer random search to grid search



"*randomly chosen trials are more efficient for hyper-parameter optimization than trials on a grid*"

Random Search for Hyper-Parameter Optimization, Bergstra and Bengio, 2012