

# Lecture 5 - Training & Regularizing Neural Networks

DD2424

March 25, 2019

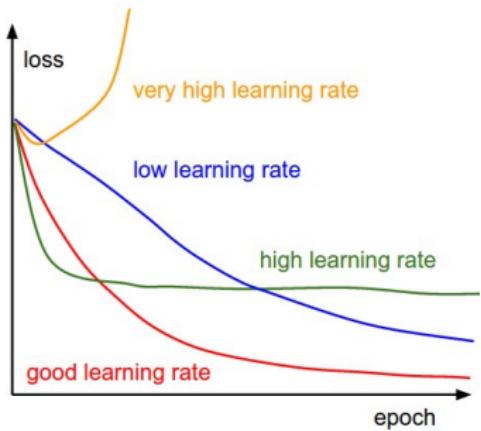
Baby sitting the training process

# Training neural networks not completely trivial!

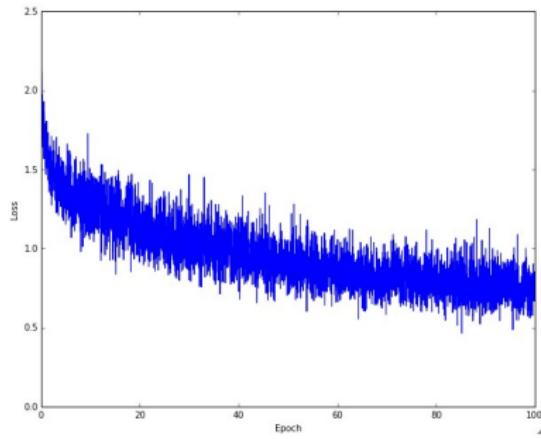
- Several **hyper-parameters** affect the quality of your training.
- These include
  - learning rate
  - degree of regularization
  - network architecture
  - hyper-parameters controlling weight initialization
- If these (potentially correlated) hyper-parameters are not appropriately set  $\implies$  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.

What to monitor during training

# Monitor & Visualize the loss/cost curve

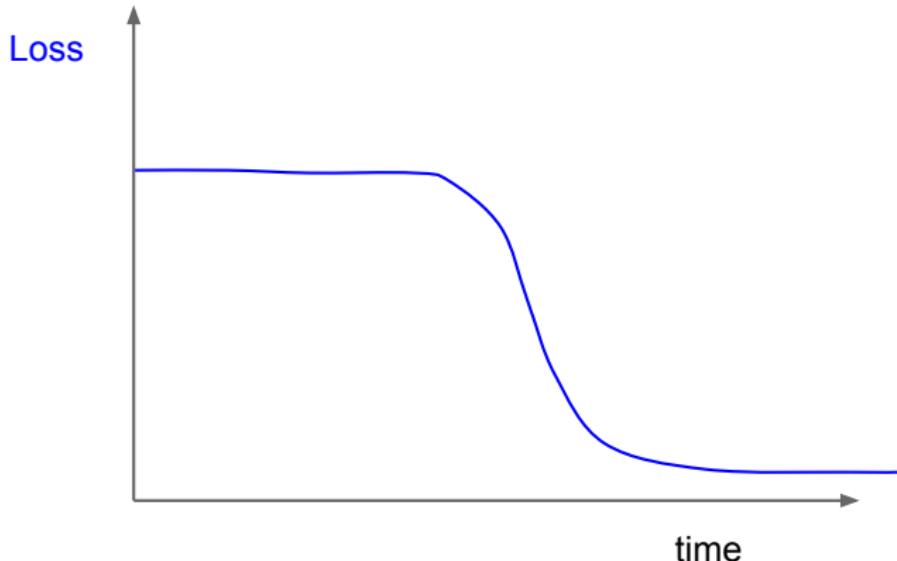


Evolution of your training loss  
is telling you something!

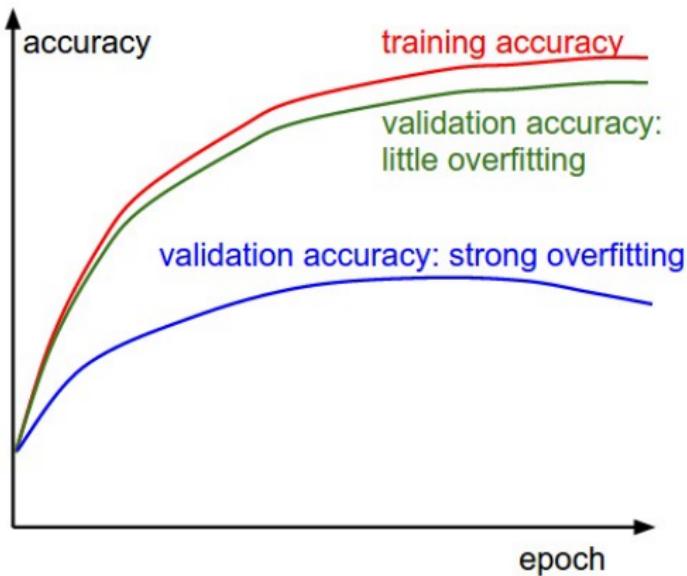


Typical training loss over time

## Telltale sign of a bad initialization



## Monitor & visualize the accuracy

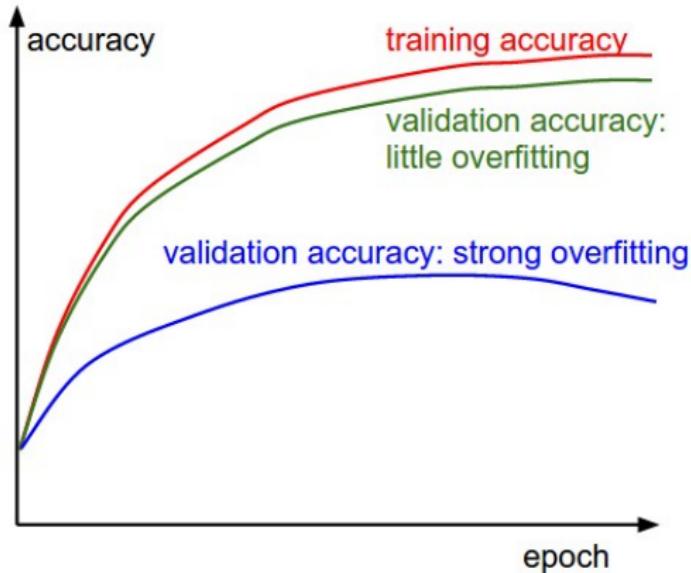


Gap between training and validation accuracy indicates amount of over-fitting.

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

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

## Monitor & visualize the accuracy



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

**Under-fitting**  $\implies$  model capacity not high enough:

- Increase the size of the network.

## Track the ratio of weight updates to weight magnitudes

- Track the **ratio** of the **magnitude of the update vector** to the **magnitude of the parameter vector**.
- So for a weight matrix,  $W$ , and vanilla SGD updates:

$$r = \frac{\| -\eta \nabla_W J \|}{\| W \|}$$

- A rough heuristic is that  $r \sim .001$ .
- If  $r \ll .001 \implies$  learning rate might be too low.
- If  $r \gg .001 \implies$  learning rate might be too high.

**Parameter Updates:** Variations of Stochastic Gradient Descent/Gradient Descent

## Remember Gradient Descent optimization

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

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

- How to choose  $\eta_t$  ?
- Depends on
  - the shape of  $f$  and
  - how fast a convergence you want.

- **Important property:**

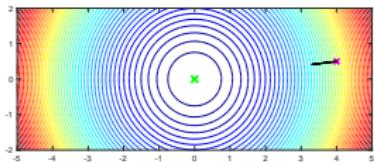
Gradient vector  $\nabla f(\mathbf{x}^{(t)})$  is always orthogonal to the iso-contour curve of  $f$  at  $\mathbf{x}^{(t)}$ .

# Example behaviour on the ideal $f$ for gradient descent

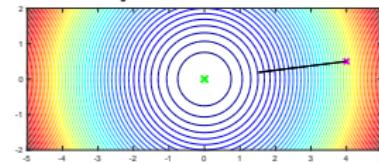
Use **gradient descent** to minimize  $f(\mathbf{x}) = x_1^2 + x_2^2$  with

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

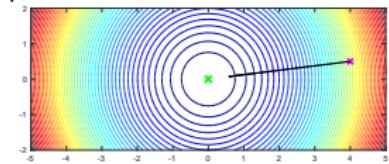
Gradient descent paths for different  $\eta_t$ 's



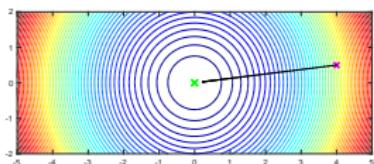
$$\eta_t = .001$$



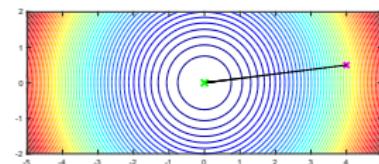
$$\eta_t = .005$$



$$\eta_t = .01$$



$$\eta_t = .015$$



$$\eta_t = .02$$

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

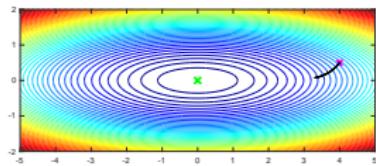
⇒ can increase  $\eta_t$  liberally (without worrying about divergence) to speed up convergence.

# Example behaviour on a suitable $f$ for gradient descent

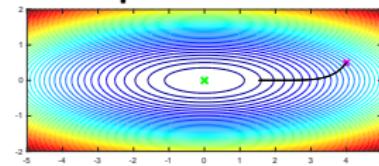
Use gradient descent to minimize  $f(\mathbf{x}) = x_1^2 + 10x_2^2$  with

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

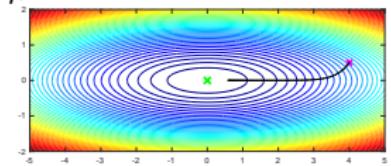
Gradient descent paths for different  $\eta_t$ 's



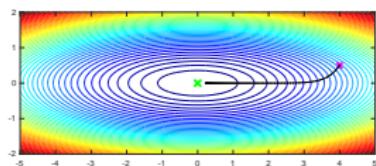
$$\eta_t = .001$$



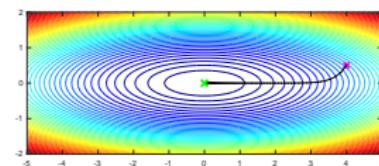
$$\eta_t = .005$$



$$\eta_t = .01$$



$$\eta_t = .015$$



$$\eta_t = .02$$

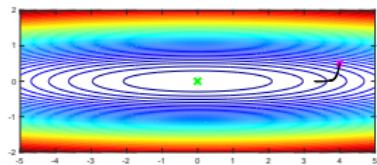
Note in this example the update direction points directly to the optimum when  $x_{t,2} = 0$ .

# Example behaviour on a less suitable $f$ for gradient descent

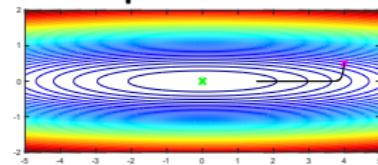
Use gradient descent to minimize  $f(\mathbf{x}) = x_1^2 + 50x_2^2$  with

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

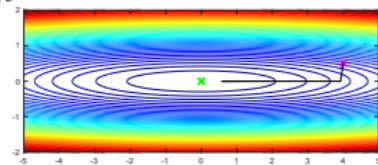
Gradient descent path for different  $\eta_t$ 's



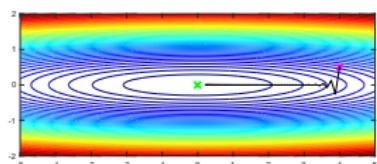
$$\eta_t = .001$$



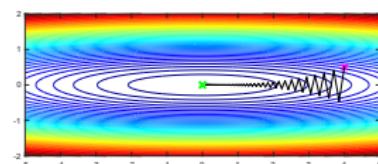
$$\eta_t = .005$$



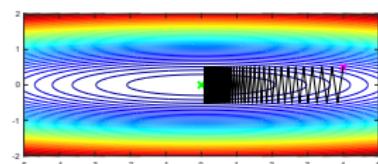
$$\eta_t = .01$$



$$\eta_t = .015$$



$$\eta_t = .019$$



$$\eta_t = .02$$

Note as  $\eta_t$  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

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 obviously  $\mathbf{x}^* = \mathbf{0}$ .)

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 obviously  $\mathbf{x}^* = \mathbf{0}$ .)

## How does GD work on this problem?

- Apply GD with fixed  $\eta$  and initial guess  $\mathbf{x}^{(0)}$  for  $\mathbf{x}^*$
- Questions:
  - For what values of  $\eta$  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 ( $\mathbf{x}^{(t)} \rightarrow \mathbf{x}^*$  at  $t \rightarrow \infty$ ) need

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

$\implies$

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

# $\eta$ that gives fastest convergence?

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

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

- 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

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

$\eta$  that gives fastest convergence?

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

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

- 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

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

$\eta$  that gives fastest convergence?

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

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

- 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

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

$\eta$  that gives fastest convergence?

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

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

- 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 $\eta^*$

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

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

- For the optimal learning rate have

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

- Let  $\kappa = \lambda_2 / \lambda_1$  then rate of convergence is defined by

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

- Larger  $\kappa$  (assuming  $\lambda_2 > \lambda_1$ )  $\implies$  closer  $(\kappa - 1) / (\kappa + 1)$  is to 1  
 $\implies$  slower the convergence of the GD estimate.

## Rate of convergence at $\eta^*$

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

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

- For the optimal learning rate have

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

- Let  $\kappa = \lambda_2 / \lambda_1$  then rate of convergence is defined by

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

- Larger  $\kappa$  (assuming  $\lambda_2 > \lambda_1$ )  $\implies$  closer  $(\kappa - 1) / (\kappa + 1)$  is to 1  
 $\implies$  slower the convergence of the GD estimate.

## Rate of convergence at $\eta^*$

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

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

- For the optimal learning rate have

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

- Let  $\kappa = \lambda_2 / \lambda_1$  then rate of convergence is defined by

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

- Larger  $\kappa$  (assuming  $\lambda_2 > \lambda_1$ )  $\implies$  closer  $(\kappa - 1) / (\kappa + 1)$  is to 1  
 $\implies$  slower the convergence of the GD estimate.

## Rate of convergence at $\eta^*$

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

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

- For the optimal learning rate have

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

- Let  $\kappa = \lambda_2 / \lambda_1$  then rate of convergence is defined by

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

- Larger  $\kappa$  (assuming  $\lambda_2 > \lambda_1$ )  $\implies$  closer  $(\kappa - 1) / (\kappa + 1)$  is to 1  
 $\implies$  slower the convergence of the GD estimate.

Rate of convergence depends on ratio -  $\lambda_2/\lambda_1$

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

where

$$\kappa = \frac{\lambda_2}{\lambda_1}$$

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

# Can extend result to $d$ dimensions

Have

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

- GD converges when

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

- Have fastest convergence when

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

- Then for optimal  $\eta^*$

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

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

- When  $\kappa$  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  
     $\Rightarrow$  slow convergence.

**Unfortunately, in neural network cost functions**  
**ravines are common around local optima.**

## Gradient descent and the problem of choosing $\eta_t$

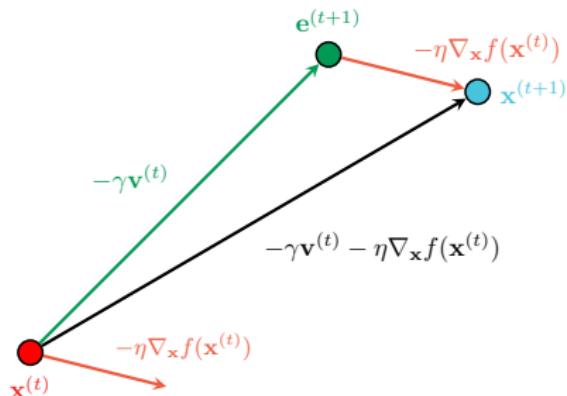
- $\eta_t$  too small  $\implies$  slow convergence speed but smooth update path.
- $\eta_t$  too large  $\implies$  optimization can potentially diverge and potentially have an inefficient zig-zag update path.

# Solution 1: SGD with momentum

- Introduce **momentum** vector as well as the gradient vector.
- Let  $\gamma \in [0, 1]$  and  $\mathbf{v}$  is the momentum vector

$$\begin{aligned}\mathbf{v}^{(t+1)} &= \gamma \mathbf{v}^{(t)} + \eta \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)}) && \leftarrow \text{update vector} \\ \mathbf{x}^{(t+1)} &= \mathbf{x}^{(t)} - \mathbf{v}^{(t+1)}\end{aligned}$$

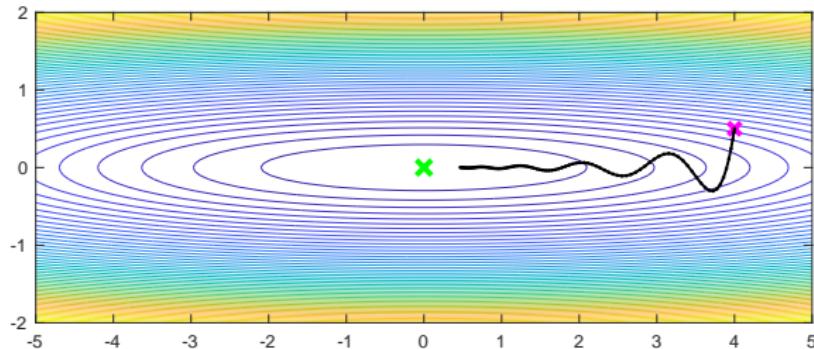
Typically  $\gamma$  set to .9.



# How and why momentum helps

How?

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



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

Why?

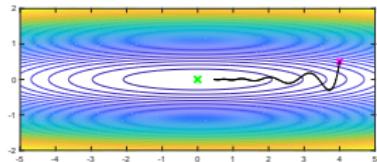
- For dimensions whose gradient is constantly changing then their entries in the update vector are damped.

# Example behaviour of “GD + mom” for a ravine

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

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

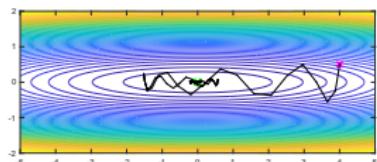
**GD + mom path for different  $\eta_t$ 's and  $\gamma = .9$**



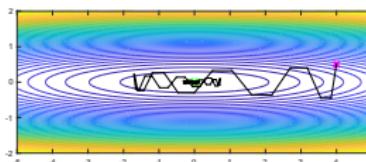
$$\eta_t = .001$$

$$\eta_t = .005$$

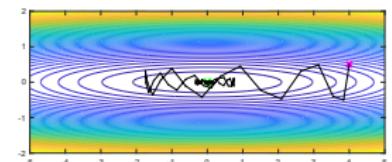
$$\eta_t = .01$$



$$\eta_t = .015$$



$$\eta_t = .019$$



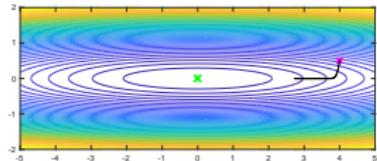
$$\eta_t = .02$$

# Example behaviour of “GD + mom” for a ravine

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

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

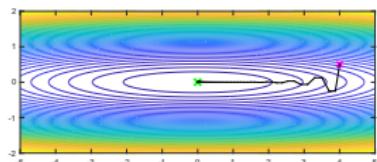
**GD + mom path for different  $\eta_t$ 's and  $\gamma = .5$**



$$\eta_t = .001$$

$$\eta_t = .005$$

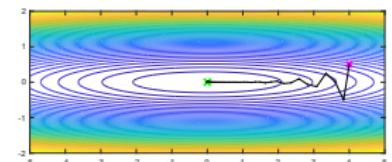
$$\eta_t = .01$$



$$\eta_t = .015$$

$$\eta_t = .019$$

$$\eta_t = .02$$



## Can view momentum as GD with a variable learning rate

- The gradient descent at update  $t$  can be written as

$$\mathbf{x}^{(t)} = \mathbf{x}^{(0)} - \eta \sum_{i=0}^{t-1} \nabla_{\mathbf{x}} f(\mathbf{x}^{(i)})$$

- The gradient descent + momentum at update  $t$  can be written as

$$\mathbf{x}^{(t)} = \mathbf{x}^{(0)} - \eta \sum_{i=0}^{t-1} \left( \frac{1 - \gamma^{t-i}}{1 - \gamma} \right) \nabla_{\mathbf{x}} f(\mathbf{x}^{(i)})$$

Have only scratched the surface here...

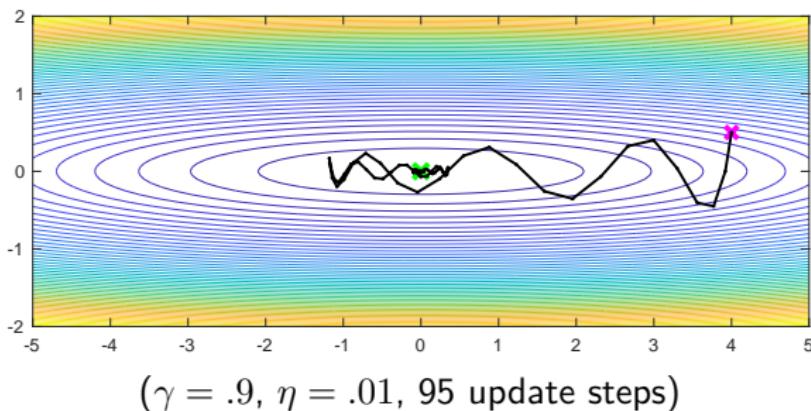
For a more sophisticated analysis and clear exposition of momentum please check out: [Why Momentum Really Works](#)

# Momentum not the complete answer

- When using momentum

⇒ can pick up too much speed in one direction.

⇒ can overshoot the local optimum.



# Solution: Nesterov accelerated gradient (NAG)

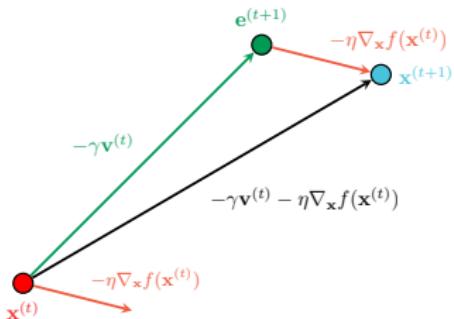
- Look and measure ahead.
- Use gradient at an estimate of the parameters at the next iteration.
- Let  $\gamma \in [0, 1]$  then

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

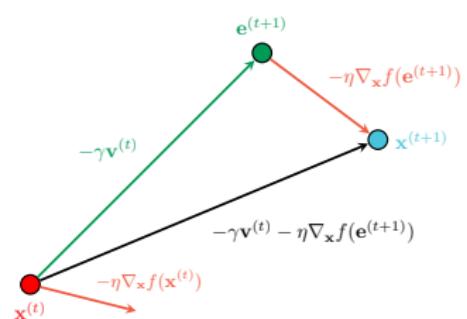
$$\mathbf{v}^{(t+1)} = \gamma \mathbf{v}^{(t)} + \eta \nabla_{\mathbf{x}} f(\mathbf{e}^{(t+1)}) \quad \leftarrow \text{update vector}$$

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

Typically  $\gamma$  set to .9.



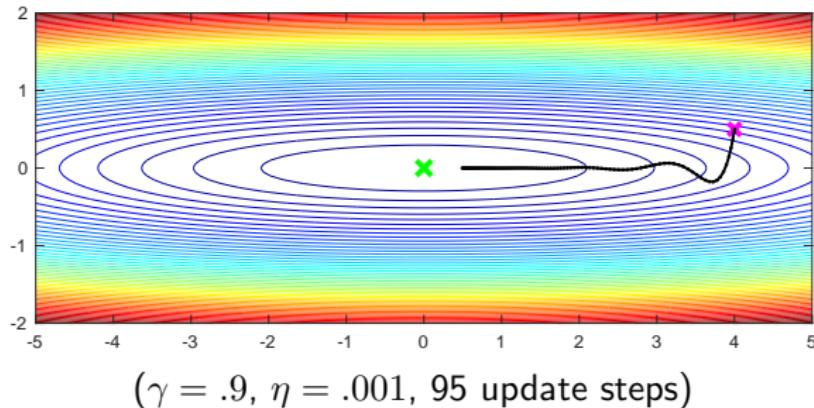
Momentum update



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  $f$ .



## Note:

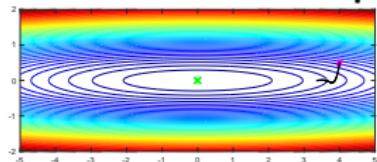
NAG shown to greatly increase the ability to train RNNs:

# Example behaviour of “NAG” for a ravine

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

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

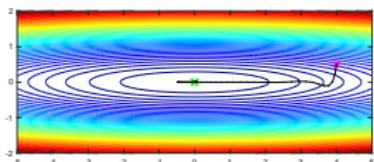
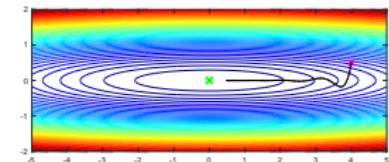
**NAG path for different  $\eta_t$ 's and  $\gamma = .9$**



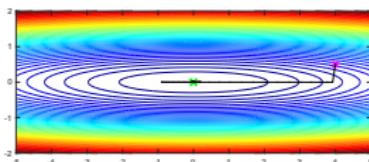
$$\eta_t = .0001$$

$$\eta_t = .0005$$

$$\eta_t = .001$$



$$\eta_t = .005$$



$$\eta_t = .01$$

# More convenient form of NAG update

- Let  $\gamma \in [0, 1]$  then

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

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

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

- Form of update inconvenient as usually have  $\mathbf{x}^{(t)}, \nabla_{\mathbf{x}} f(\mathbf{x}^{(t)})$ .

- Can make a variable transformation

$$\mathbf{m}^{(t)} = \mathbf{x}^{(t)} - \gamma \mathbf{v}^{(t)} \quad (\equiv \mathbf{e}^{(t+1)})$$

- Can write  $\mathbf{x}^{(t+1)}$  as

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

$\implies$

$$\begin{aligned}\mathbf{m}^{(t+1)} &= \mathbf{x}^{(t)} - (1 + \gamma) \mathbf{v}^{(t+1)} \\ &= \mathbf{m}^{(t)} + \gamma \mathbf{v}^{(t)} - (1 + \gamma) \mathbf{v}^{(t+1)}\end{aligned}$$

where

$$\mathbf{v}^{(t+1)} = \gamma \mathbf{v}^{(t)} + \eta \nabla_{\mathbf{x}} f(\mathbf{m}^{(t)})$$

- 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

- For a cleaner statement introduce some notation:

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

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

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

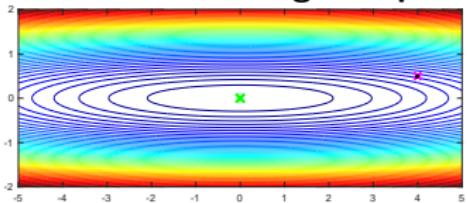
- The AdaGrad update step for each dimension is

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

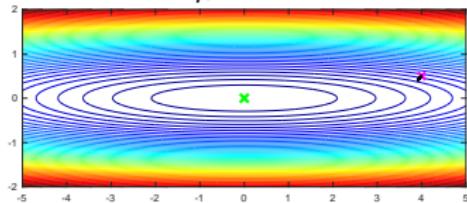
- Usually set  $\epsilon = 1e-8$  and  $\eta = .01$ .

# Adagrad's performance on the ravine

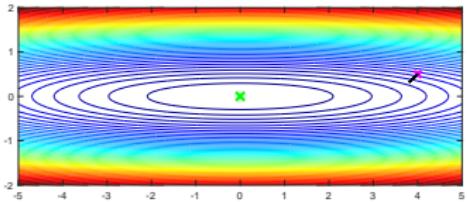
Adagrad's paths for different  $\eta_t$ 's



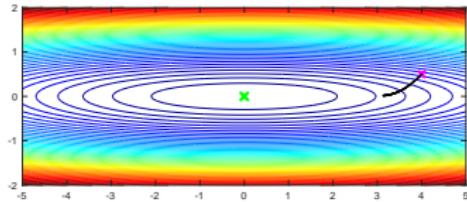
$\eta_t = .001$



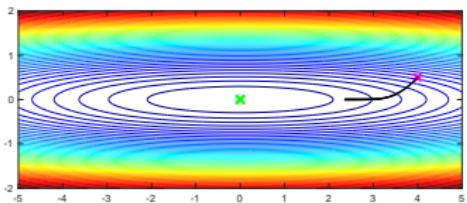
$\eta_t = .005$



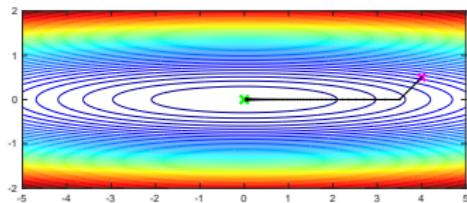
$\eta_t = .01$



$\eta_t = .05$



$\eta_t = .1$



$\eta_t = .5$

## Big weakness of AdaGrad

- Each  $g_{t,i}^2$  is positive.
  - ⇒ Each  $G_{t,i} = \sum_{j=1}^t g_{j,i}^2$  keeps growing during training.
  - ⇒ the effective learning rate  $\eta / (\sqrt{G_{t,i} + \epsilon})$  shrinks and eventually  $\rightarrow 0$ .
  - ⇒ updates of  $\mathbf{x}^{(t)}$  stop.

- 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 the update vector

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

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

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

- The AdaDelta update step:

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

Also addresses AdaGrad's radically diminishing learning rate:

- RMSProp is 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:

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

- The RMSProp update rule:

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

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

# 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 a lot and
    - ★ increase the update in dimensions with low variation.

## Update equations for Adam

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

$$\mathbf{m}^{(t+1)} = \beta_1 \mathbf{m}^{(t)} + (1 - \beta_1) \mathbf{g}_t$$

$$\mathbf{v}^{(t+1)} = \beta_2 \mathbf{v}^{(t)} + (1 - \beta_2) \mathbf{g}_t \cdot * \mathbf{g}_t$$

- 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:

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

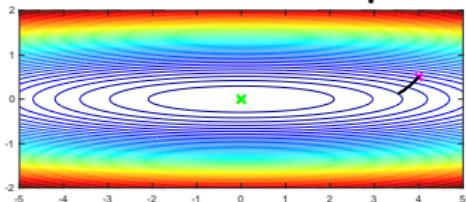
- The Adam update rule:

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

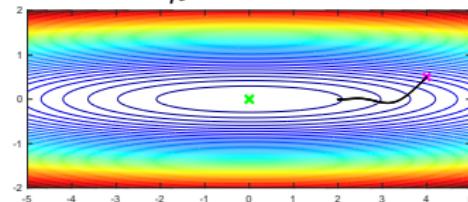
- Suggested default values  $\beta_1 = .9, \beta_2 = .999, \epsilon = 10^{-8}$ .

# Adam's performance on the ravine

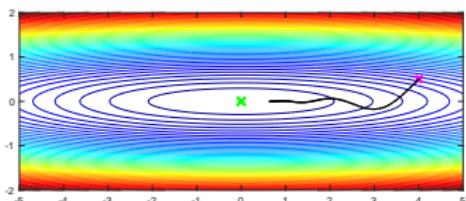
**Adam paths for different  $\eta_t$ 's**



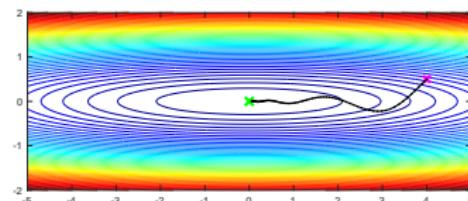
$\eta_t = .001$



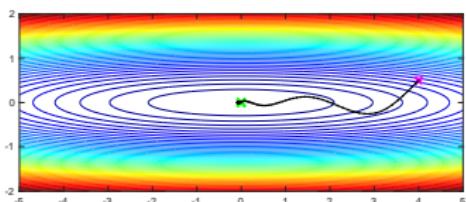
$\eta_t = .005$



$\eta_t = .01$



$\eta_t = .015$



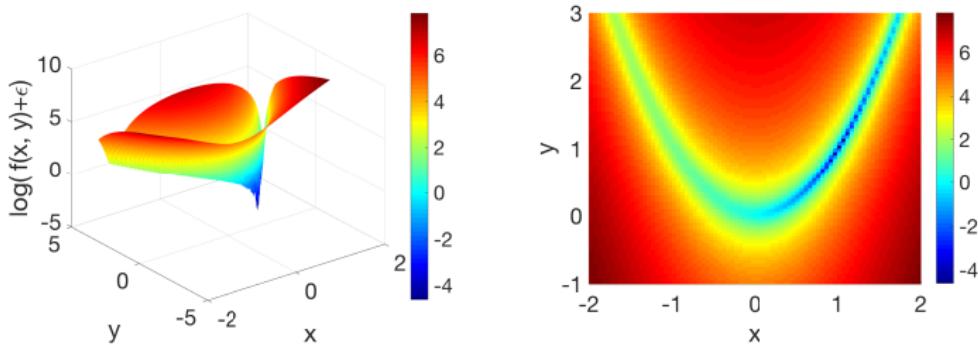
$\eta_t = .02$

# Comparison of performance on the Rosenbrock function

- 2D Rosenbrock function

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

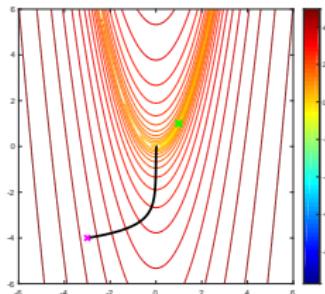
- Visualization of  $\log(f(\mathbf{x}) + \epsilon)$



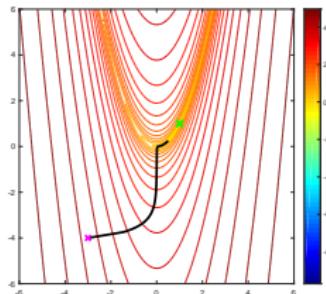
top-down viewpoint

## Rosenbrock optimization with GD

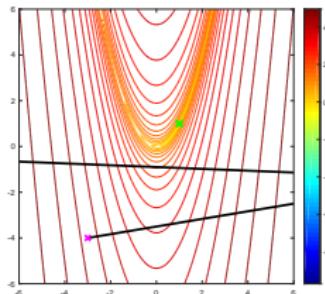
Learning rate:  $\eta_0 = .00001$  and # of update steps:  $n_{\text{iter}} = 5000$ .



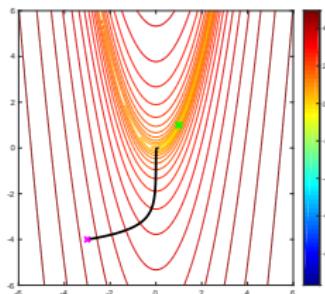
$(\eta_0, n_{\text{iter}})$



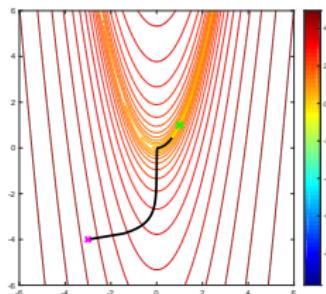
$(10\eta_0, n_{\text{iter}})$



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



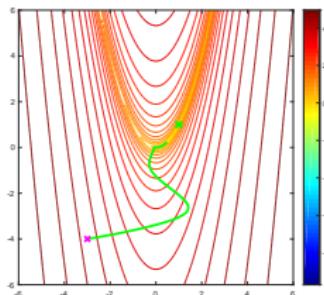
$$(\eta_0, 2n_{\text{iter}})$$



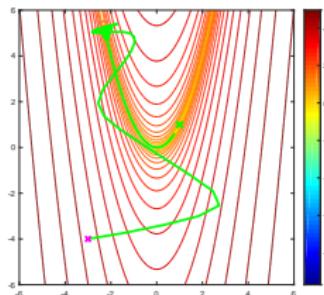
$$(10\eta_0, 2n_{\text{iter}})$$

# Rosenbrock optimization with GD+Momentum

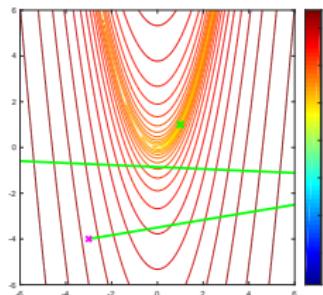
Learning rate:  $\eta_0 = .00001$  and # of update steps:  $n_{\text{iter}} = 5000$ ,  $\gamma = .9$ .



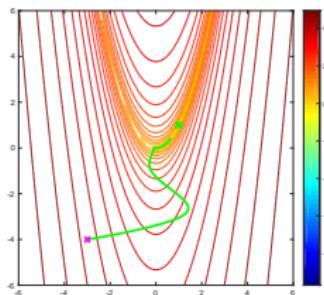
$(\eta_0, n_{\text{iter}})$



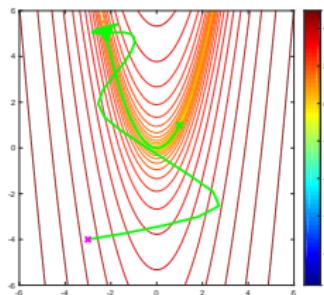
$(10\eta_0, n_{\text{iter}})$



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



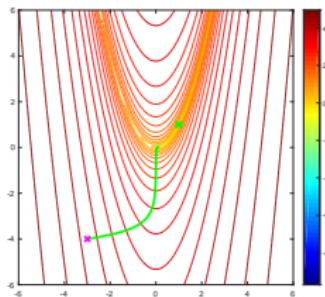
$(\eta_0, 2n_{\text{iter}})$



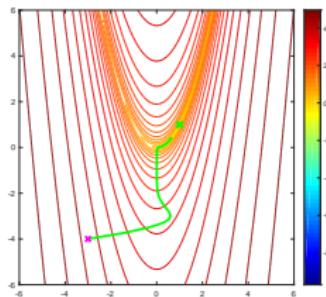
$(10\eta_0, 2n_{\text{iter}})$

# Rosenbrock optimization with GD+Momentum

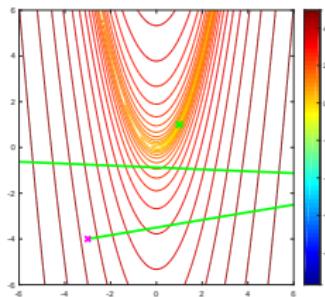
Learning rate:  $\eta_0 = .00001$  and # of update steps:  $n_{\text{iter}} = 5000$ ,  $\gamma = .5$ .



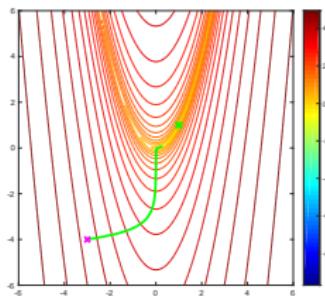
$(\eta_0, n_{\text{iter}})$



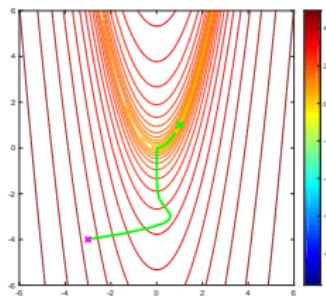
$(10\eta_0, n_{\text{iter}})$



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



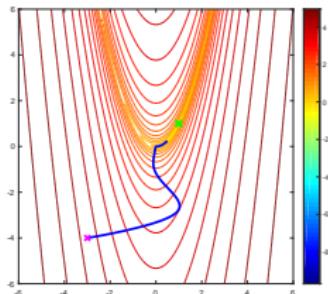
$(\eta_0, 2n_{\text{iter}})$



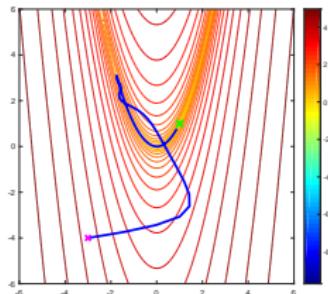
$(10\eta_0, 2n_{\text{iter}})$

# Rosenbrock optimization with NAG

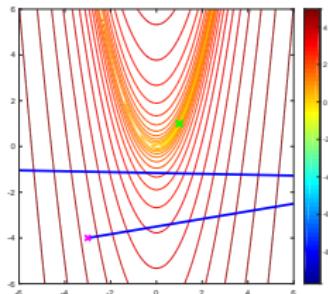
Learning rate:  $\eta_0 = .00001$  and # of update steps:  $n_{\text{iter}} = 5000$ ,  $\gamma = .9$ .



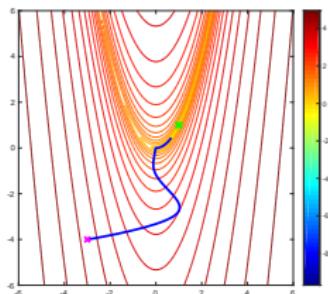
$(\eta_0, n_{\text{iter}})$



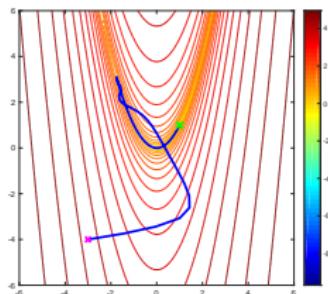
$(10\eta_0, n_{\text{iter}})$



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



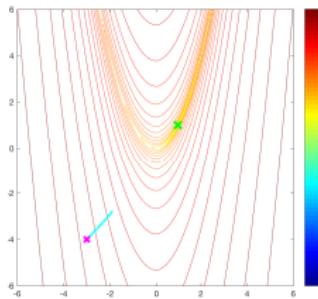
$(\eta_0, 2n_{\text{iter}})$



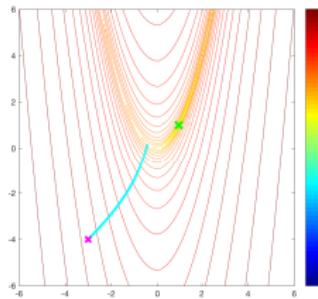
$(10\eta_0, 2n_{\text{iter}})$

# Rosenbrock optimization with Adagrad

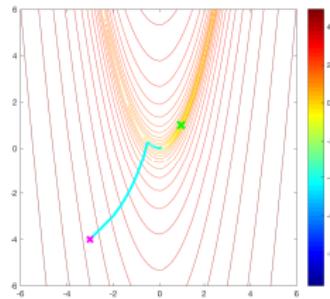
Learning rate:  $\eta_0 = .01$  and # of update steps:  $n_{\text{iter}} = 5000$ .



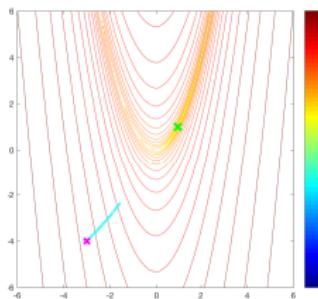
$(\eta_0, n_{\text{iter}})$



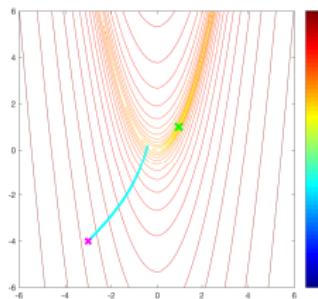
$(10\eta_0, n_{\text{iter}})$



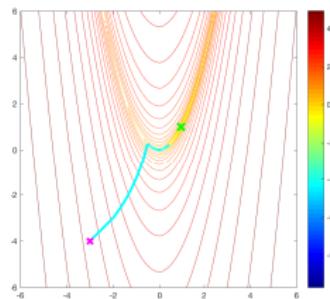
$(100\eta_0, n_{\text{iter}})$



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



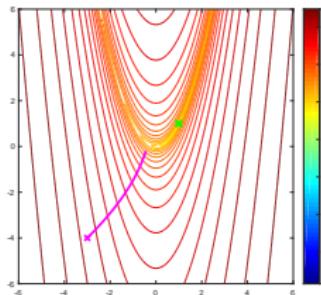
$(10\eta_0, 2n_{\text{iter}})$



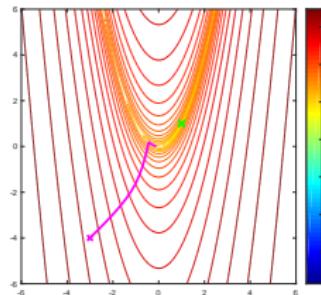
$(100\eta_0, 2n_{\text{iter}})$

## Rosenbrock optimization with Adam

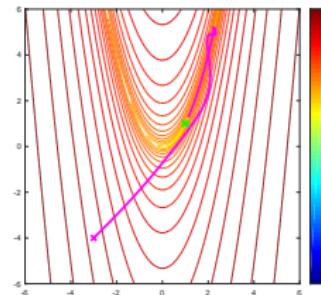
Learning rate:  $\eta_0 = .001$  and # of update steps:  $n_{\text{iter}} = 5000$ .



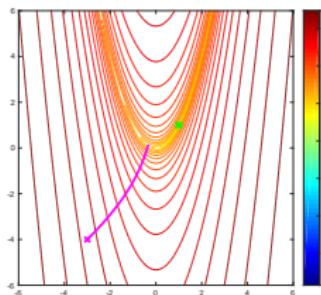
$(\eta_0, n_{\text{iter}})$



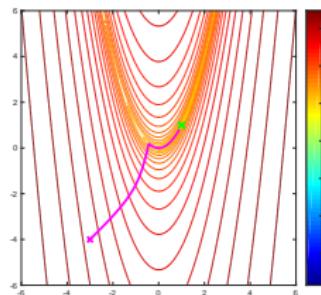
$(10\eta_0, n_{\text{iter}})$



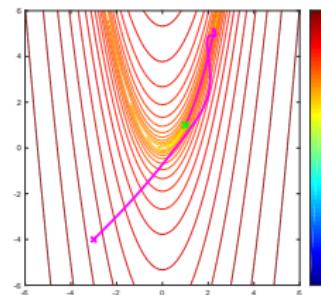
$(100\eta_0, n_{\text{iter}})$



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



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



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

## Which optimizer to use?

- Data sparse  $\implies$  likely to achieve best results using one of the adaptive learning-rate methods.
- Using the adaptive learning-rate methods  $\implies$  won't need to tune the learning rate (much!).
- RMSprop, AdaDelta, and Adam are very similar algorithms that do well in similar circumstances.
- Adam slightly outperforms RMSProp near the end of optimization.
- Adam might be the best overall choice.
- But vanilla SGD (without momentum) and a simple learning rate annealing schedule may be sufficient. But time until finding a local minimum may be long....

Annealing the learning rate

## Useful 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.
  - $\Rightarrow$  can reach into deeper, but narrower parts of the loss function.
- But knowing when to decay the learning rate is tricky!
- Decay too slowly  $\Rightarrow$  waste computations bouncing around chaotically with little improvement.
- Decay too aggressively  $\Rightarrow$  system unable to reach the best position it can.

# Common approaches to learning rate decay

- **Step decay:**

After every  $n$ th 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.)

- **Exponential decay:**

$$\eta = \eta_0 e^{-kt}$$

where  $t$  is iteration number (either w.r.t. number of update steps or epochs). Then  $\eta_0$  and  $k$  are hyper-parameters.

- **$1/t$  decay:**

$$\eta = \frac{\eta_0}{1 + kt}$$

# Common approaches to learning rate decay

- **Step decay:**

After every  $n$ th 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.)

- **Exponential decay:**

$$\eta = \eta_0 e^{-kt}$$

where  $t$  is iteration number (either w.r.t. number of update steps or epochs). Then  $\eta_0$  and  $k$  are hyper-parameters.

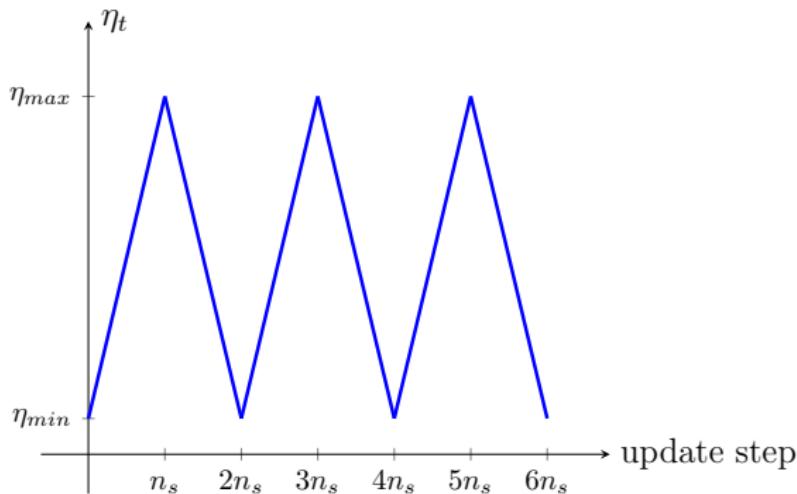
- **$1/t$  decay:**

$$\eta = \frac{\eta_0}{1 + kt}$$

Step decay most common. Better to decay conservatively and train for longer.

**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](#).

Optimization of the training hyper-parameters

## Hyperparameters to adjust

- Initial learning rate.
- Learning rate decay schedule.
- Regularization strength
  - $L_2$  penalty
  - Dropout strength

## Cross-validation strategy

- Do a **coarse** → **fine** cross-validation in stages.
- **Stage 0:** Identify the range of feasible learning rates & regularization penalties. (usually done interactively and train only for a few updates.)
- **Stage 1:** Broad search. Goal is to narrow the search range.  
Only run training for a few epochs.
- **Stage 2:** Finer search. Increase training times.
- **Stage ...:** Repeat Stage 2 as necessary.

Use performance on the **validation set** to identify good hyper-parameter settings.

# Hyper-parameter ranges

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

**Example:**

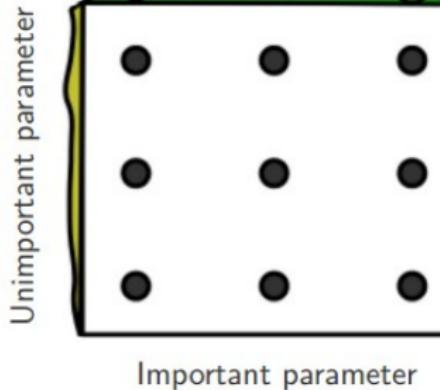
Generate a potential learning rate with

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

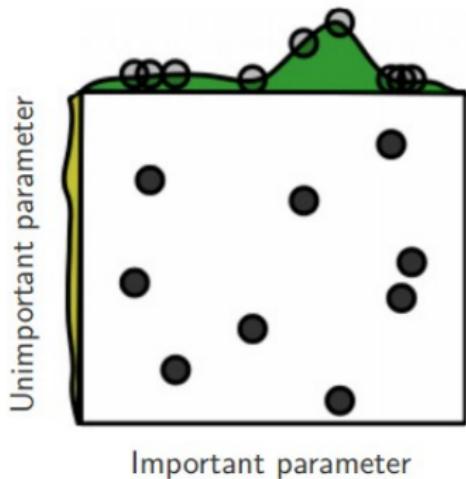
$$\eta = 10^\alpha$$

# Prefer random search to grid search

Grid Layout



Random Layout



*"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

## Evaluation: Model Ensembles

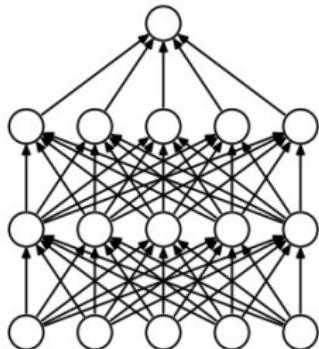
- Train multiple independent models (same hyper-parameter settings, different initializations). ( $\sim 5$  models)
- At test time apply each model and average their results.

## Model Ensemble on the cheap

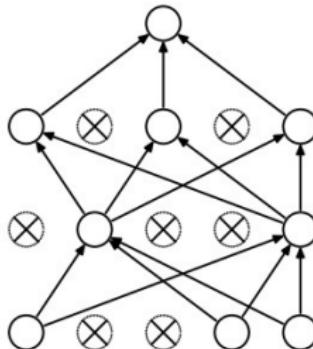
- Can also get a small boost from averaging multiple model checkpoints of a single model.
- At test time apply each model and average their results.

## Regularization Via **Dropout**

- Randomly set some activations to zero in forward pass



(a) Standard Neural Net



(b) After applying dropout.

[Srivastava et al.]

- Training practice introduced by Hinton.
- **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 2-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 switch off

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

$$\mathbf{x}^{(1)} = \mathbf{x}^{(1)}. * \mathbf{u}_1$$

3. Repeat the process for the next layer

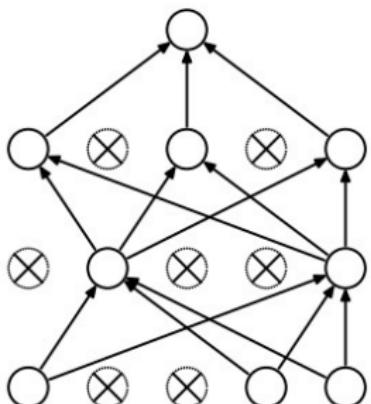
$$\mathbf{x}^{(2)} = \max(0, W_2 \mathbf{x}^{(1)} + \mathbf{b}_2)$$

$$\mathbf{u}_2 = \text{rand}(\text{size}(\mathbf{x}^{(2)})) < p \quad (\text{Matlab notation})$$

$$\mathbf{x}^{(2)} = \mathbf{x}^{(2)}. * \mathbf{u}_2$$

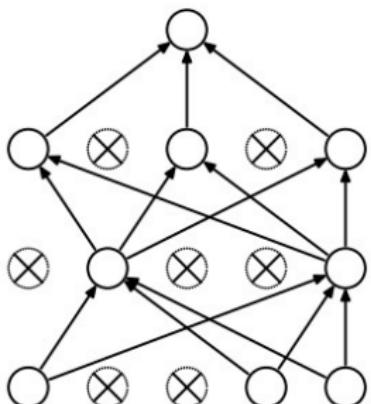
4. Output:  $\mathbf{x}^{(3)} = \text{SoftMax}(W_3 \mathbf{x}^{(2)} + \mathbf{b}_3)$

# Why is this a good idea?

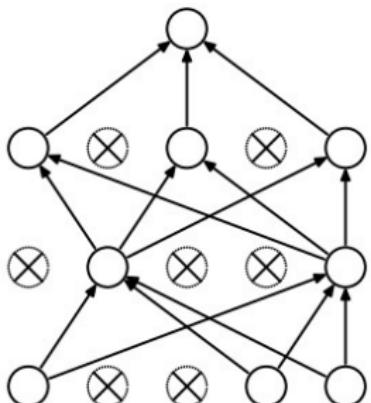


- Forces the network to have a redundant representation.
- Another interpretation
  - Dropout is training a large ensemble of models.
  - Each binary mask is one model, gets trained on only  $\sim$ one datapoint.

# Why is this a good idea?



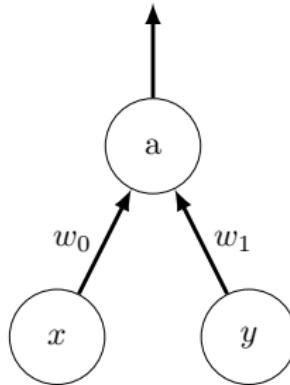
- Forces the network to have a redundant representation.
- **Another interpretation**
  - Dropout is training a large ensemble of models.
  - Each binary mask is one model, gets trained on only  $\sim$ one datapoint.



- **Ideally:** Want to integrate out all the noise.
- **Monte Carlo approximation**
  - Do many forward passes with different dropout masks.
  - Average all the predictions.

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

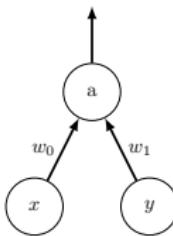
At test time



Consider this simple partial network

- During testing if we do not compensate:

$$a_{\text{test}} = w_0x + w_1y$$



Consider this simple partial network

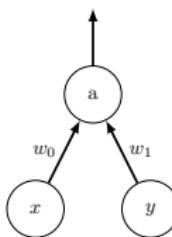
- During dropout training:

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.

$$\begin{aligned}
 E[a_{\text{training}}] &= (1 - p)^2(w_00 + w_10) + p(1 - p)(w_0x + w_10) \\
 &\quad + p(1 - p)(w_00 + w_1y) + p^2(w_0x + w_1y) \\
 &= p(w_0x + w_1y)
 \end{aligned}$$



Consider this simple partial network

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

$$a_{\text{test}} = w_0x + w_1y$$

- During **dropout training**:

$$\mathbb{E}[a_{\text{training}}] = p(w_0x + w_1y) = p a_{\text{test}}$$

⇒ 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:

$$\text{output at test time} = \text{expected output at training time}$$

- Don't drop activations but have to compensate

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

$$\mathbf{x}^{(2)} = \max(0, W_2 \mathbf{x}^{(1)} + \mathbf{b}_2) * p$$

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

## More common: Inverted Dropout

- During **training**:

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

$$\mathbf{u}_2 = (\text{rand}(\text{size}(\mathbf{x}^{(1)})) < p)/p \quad \leftarrow \text{Note } /p$$

$$\mathbf{x}^{(1)} = \mathbf{x}^{(1)} . * \mathbf{u}_2$$

$$\mathbf{x}^{(2)} = \max(0, W_2 \mathbf{x}^{(1)} + \mathbf{b}_2)$$

$$\mathbf{u}_2 = (\text{rand}(\text{size}(\mathbf{x}^{(2)})) < p)/p \quad \leftarrow \text{Note } /p$$

$$\mathbf{x}^{(2)} = \mathbf{x}^{(2)} . * \mathbf{u}_3$$

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

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

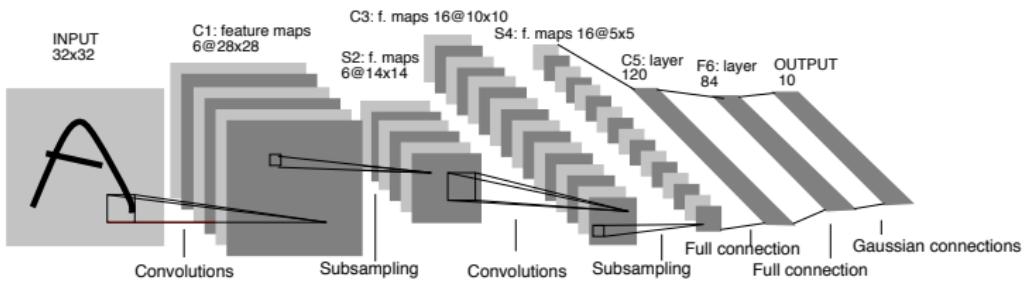
$$\mathbf{x}^{(1)} = \max(0, W_1 \mathbf{x}^{(0)} + \mathbf{b}_1)$$

$$\mathbf{x}^{(2)} = \max(0, W_2 \mathbf{x}^{(1)} + \mathbf{b}_2)$$

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

## Convolutional Neural Networks (ConvNets)

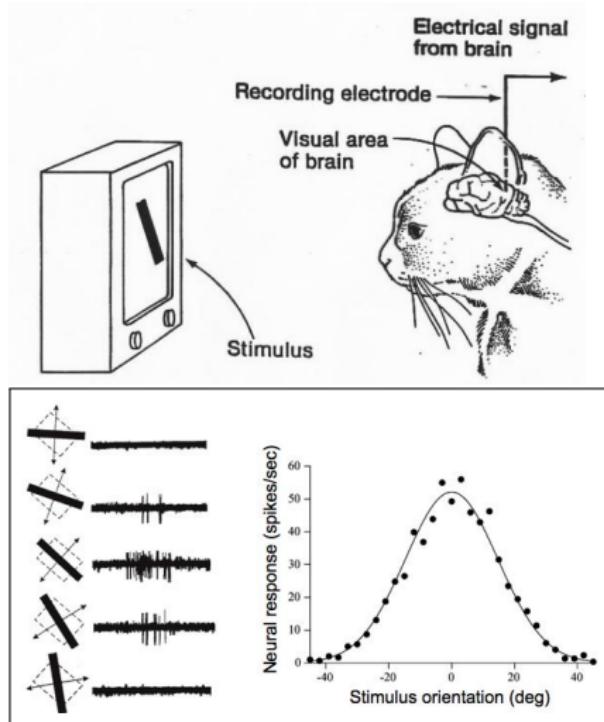
# Convolutional Neural Networks



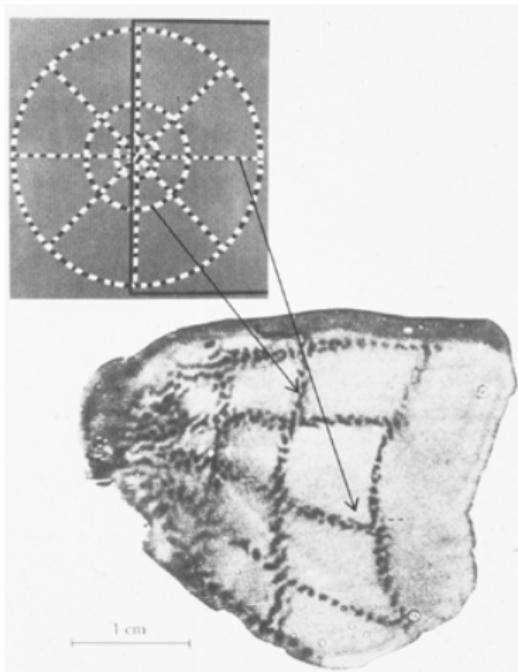
**LeNet-5 (LeCun '98)**

## ConvNets: Some history

# Hubel & Wiesel cat experiments 1968

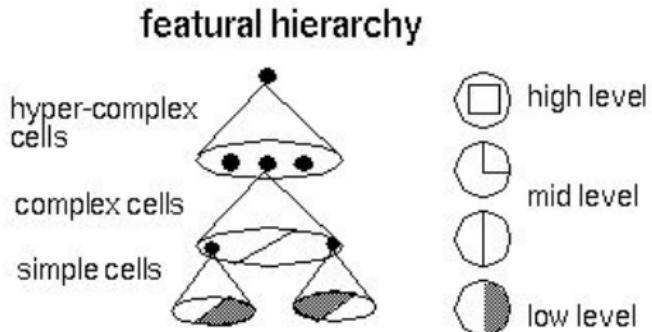
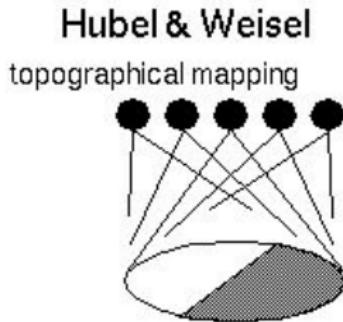


- Discovered *visual cortex* consists of a hierarchy of simple, complex, and hyper-complex cells.  
(Experiments in 50's & 60's)
- Hubel & Wiesel won the Nobel prize (1981).



**Topographical mapping in the cortex:** nearby cells in cortex represented nearby regions in the visual field.

# Hierarchical organization



**Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position** by Kunihiko Fukushima, 1980.

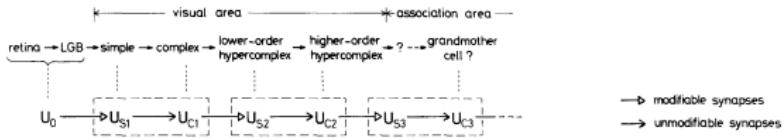
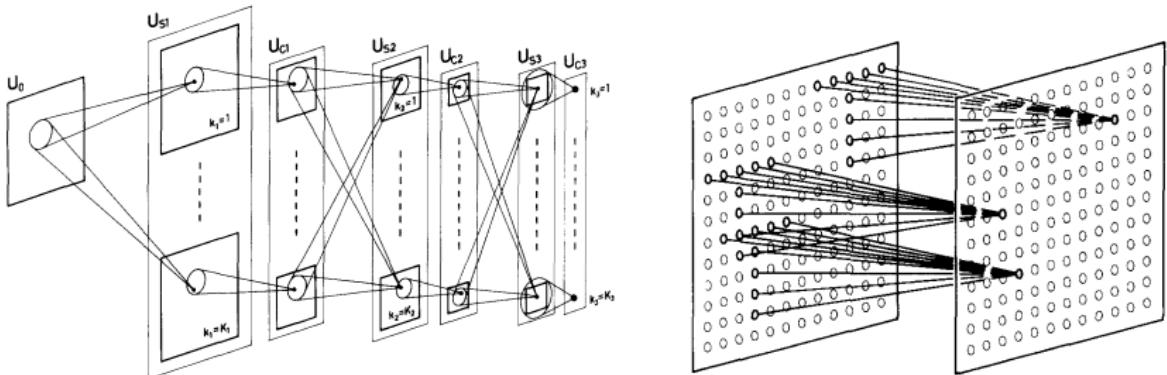


Fig. 1. Correspondence between the hierarchy model by Hubel and Wiesel, and the neural network of the neocognitron

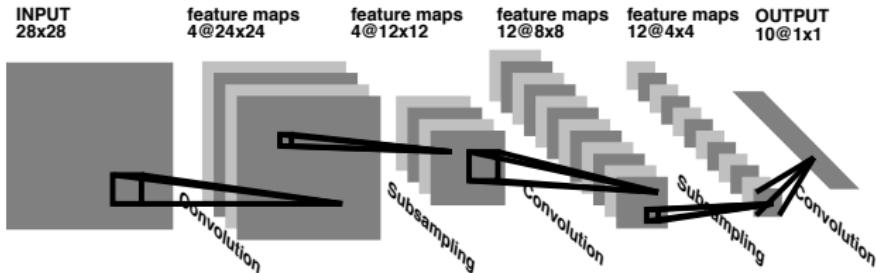
## Inspired by Hubel & Wiesel model



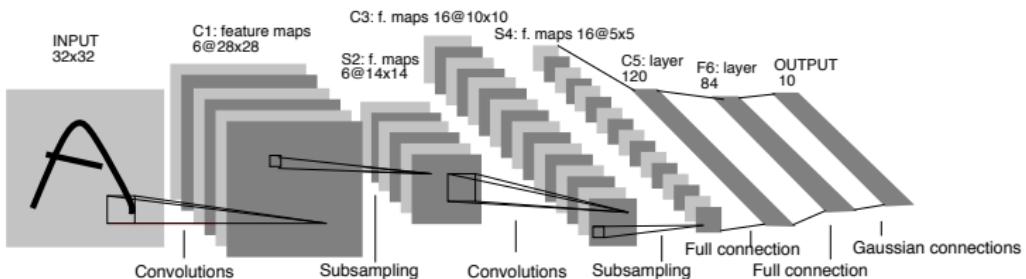
*sandwich architecture (SCSCSC...)*

**simple cells:** modifiable parameters, **complex cells:** perform pooling

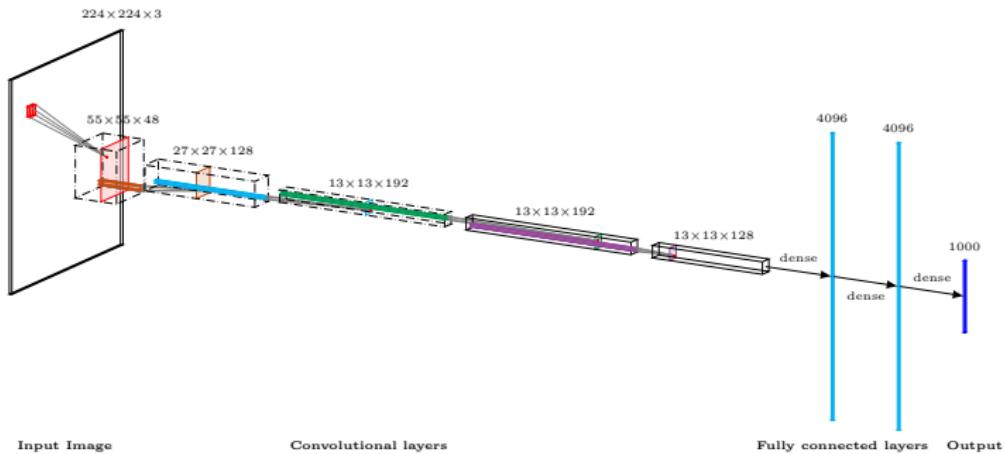
# LeCun's LeNet ConvNets



LeNet 1 '90



LeNet 5 '95



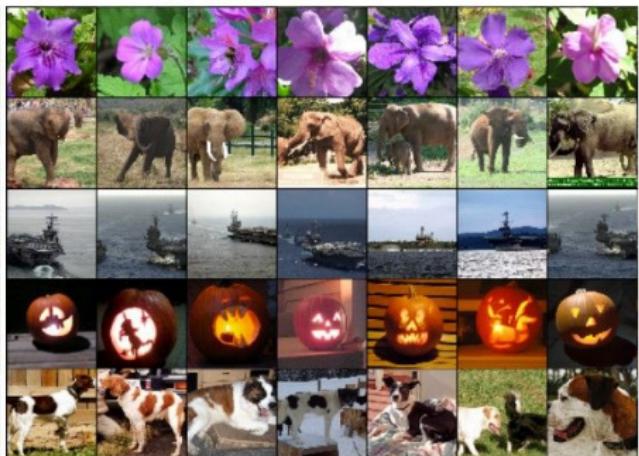
**ImageNet Classification with Deep Convolutional Neural Networks** by Krizhevsky, Sutskever, Hinton, 2012

# Fast-forward to today: ConvNets are everywhere

Classification



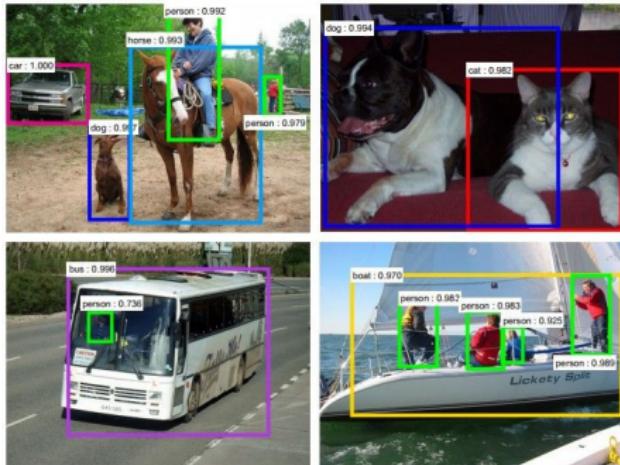
Retrieval



[Krizhevsky 2012]

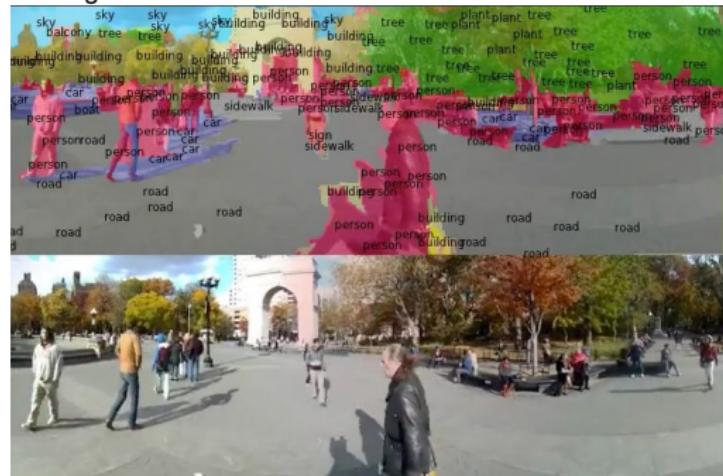
# Fast-forward to today: ConvNets are everywhere

Detection



[Faster R-CNN: Ren, He, Girshick, Sun 2015]

Segmentation



[Farabet et al., 2012]

# Fast-forward to today: ConvNets are everywhere

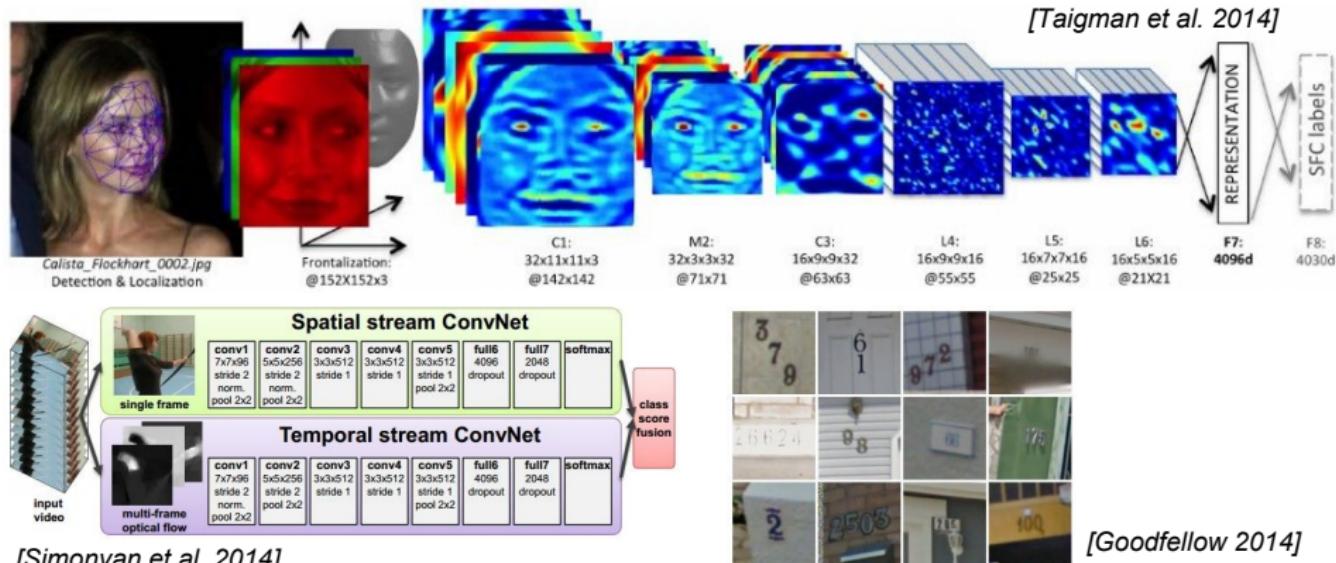


self-driving cars



NVIDIA Tegra X1

# Fast-forward to today: ConvNets are everywhere



# Fast-forward to today: ConvNets are everywhere

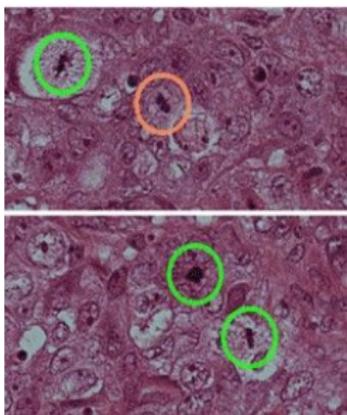


[Toshev, Szegedy 2014]



[Mnih 2013]

# Fast-forward to today: ConvNets are everywhere



[Ciresan et al. 2013]



[Sermanet et al. 2011]

[Ciresan et al.]



*Whale recognition, Kaggle Challenge*



*Mnih and Hinton, 2010*

# Image Captioning

Describes without errors



A person riding a motorcycle on a dirt road.

Describes with minor errors



Two dogs play in the grass.

Somewhat related to the image



A skateboarder does a trick on a ramp.

Unrelated to the image



A dog is jumping to catch a frisbee.



A group of young people playing a game of frisbee.



Two hockey players are fighting over the puck.



A little girl in a pink hat is blowing bubbles.



A refrigerator filled with lots of food and drinks.



A herd of elephants walking across a dry grass field.



A close up of a cat laying on a couch.



A red motorcycle parked on the side of the road.



A yellow school bus parked in a parking lot.

[Vinyals et al., 2015]