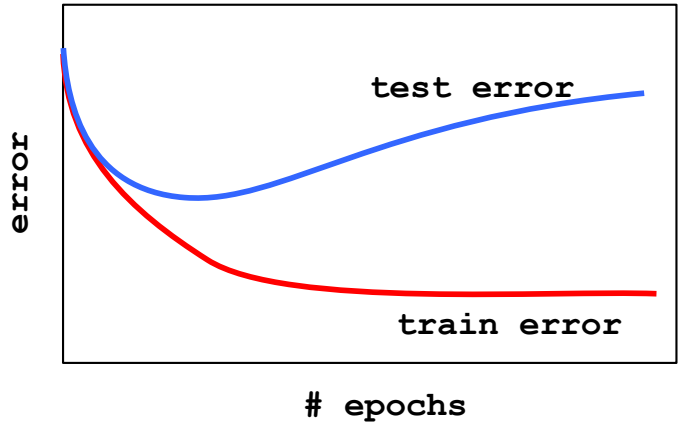
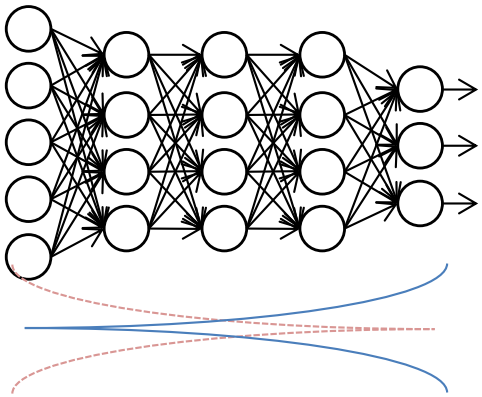
TRAINING OF NEURAL NETWORKS --- ADVANCED

Difficulty of training deep neural networks

- It is (was?) difficult to train deep networks; the deeper, the harder
- We frequently encounter overfitting
  - Your model works very well on training data but not on novel data

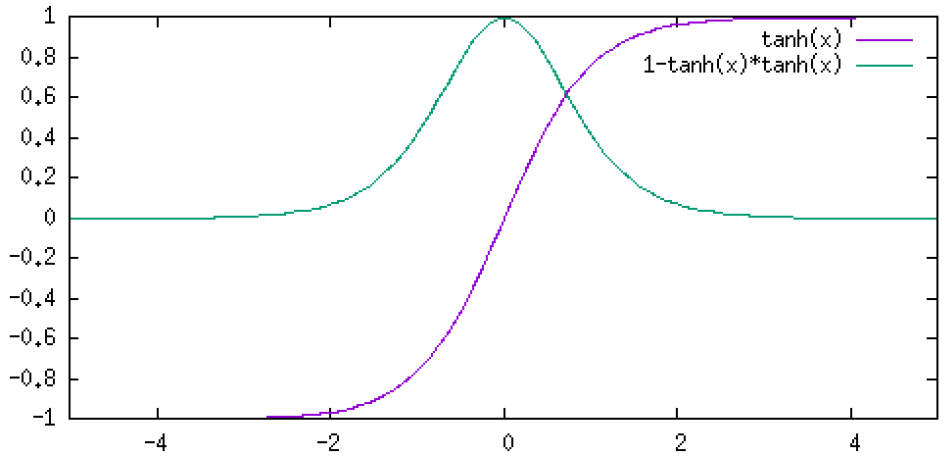


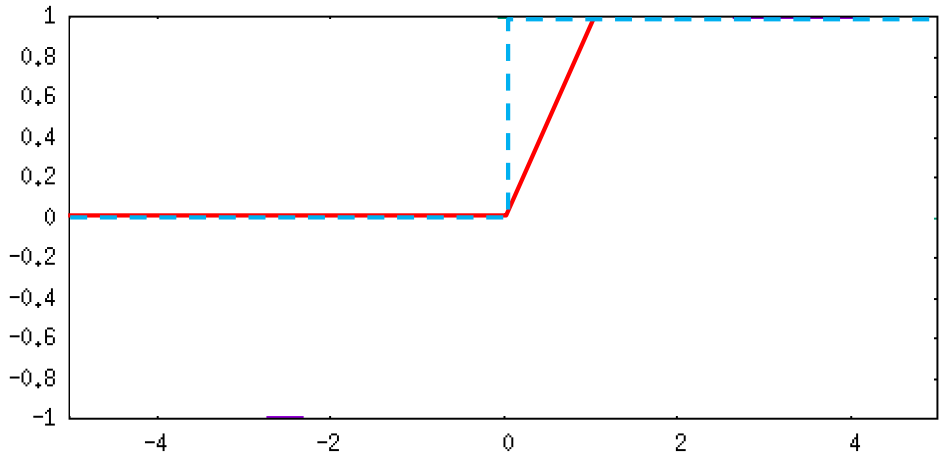
- Vanishing gradient problem
  - When backproagating them, deltas tend to blow up to infinity or shrink to zero



Why do gradients vanish?

- If traditional activation functions are employed
  - There is a bound in the output range of forward computation
  - No bound on the output range of backward computation; backprop of deltas is pure linear computation
  - Input range for which the gradient of act-funcs is nonzero is narrow
- These are not the case with ReLU
  - This is considered to be a reason why ReLU makes training easier





Weight decay: a simple regularization

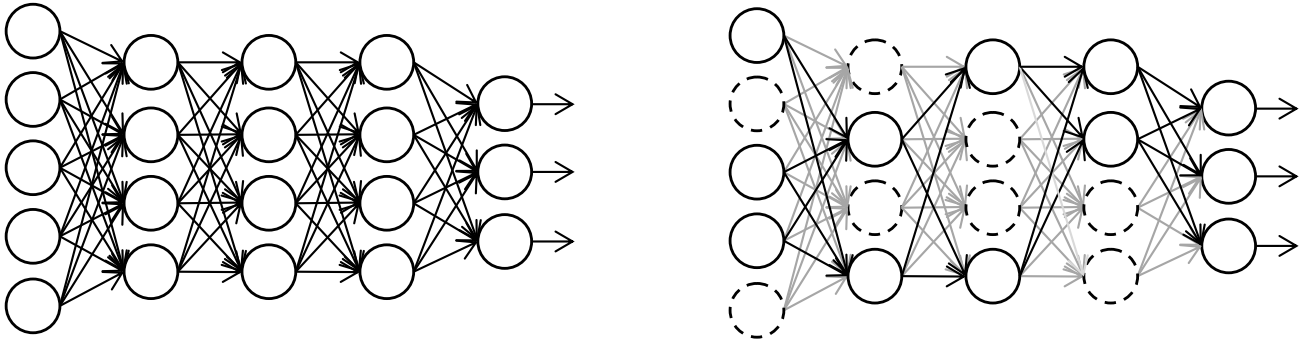
- *Regularization* to reduce degrees of freedom of weights at the training time
  - Prevent weights to blow up to infinity
  - A small value is chosen for λ

$$E_t(\mathbf{w}) \equiv \frac{1}{N_t} \sum_{n \in \mathcal{D}_t} E_n(\mathbf{w}) + \frac{\lambda}{2} \|\mathbf{w}\|^2$$

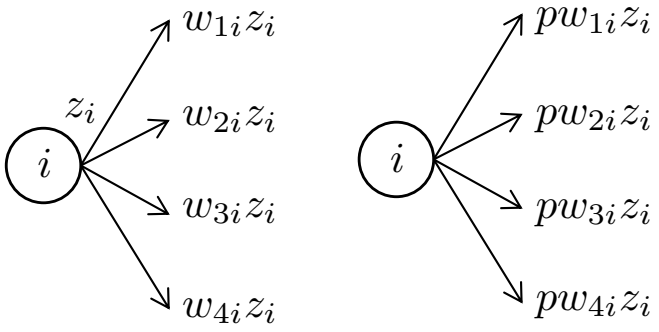
$$\mathbf{w}^{(t+1)} = \mathbf{w}^{(t)} - \epsilon \left(\frac{1}{N_t} \sum \nabla E_n + \lambda \mathbf{w}^{(t)} \right)$$

Dropout

- Randomly choose units of intermediate layer and invalidate them
  - With probability *p*; only at the training time



- Multiply unit outputs by *p* at the test time
  - To compensate for the invalidation



- Can be interpreted as regularization at the training time; the net works as an ensemble of multiple networks at the test time

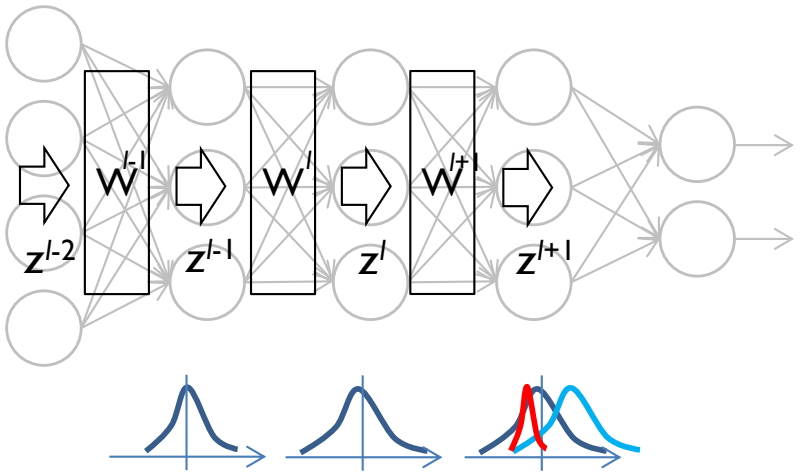
Batch normalization

- Weights are initialized so as to match distributions of layer outputs
- But this equilibrium won't last long as training proceeds
  - Weights are updated at each minibatch
- BN normalizes the layer distribution over a minibatch using its stati stics
  - Mean and stddev of layer outputs over the minibatch samples are used
  - Then normalize each layer output as

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$$

  - Provide some freedom
    w/ learnable parameters

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathbf{BN}_{\gamma,\beta}(x_i)$$



Data augmentation: basics

- The more training samples are, the better the result will be
  - However, hard to get many samples with true labels
- We create 'new' samples from existing ones = data augmentation
- We can use all sorts of image transformation that do not change 'contents'
  - Crop, horizontal/vertical flip
  - Geometric warp: scaling, sheer, etc.
  - Color changes

[Karianakis+2015]

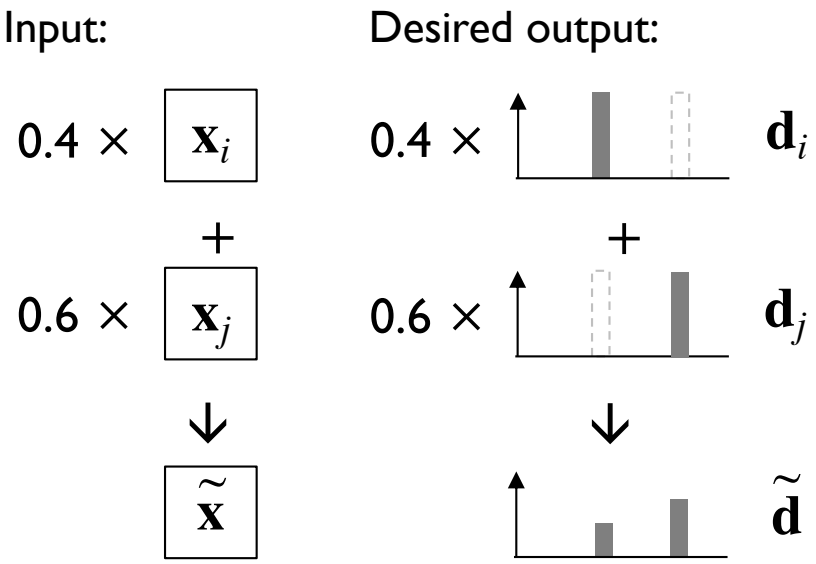
Data augmentation: advanced

- Cutout [Devries+2017]
  - Mask a part of the image with randomly generated gray square
  - Train the model to predict the class correctly

- Mixup [Zhang+2018]
  - Pick two samples w/ different labels and *mix up* them as follows

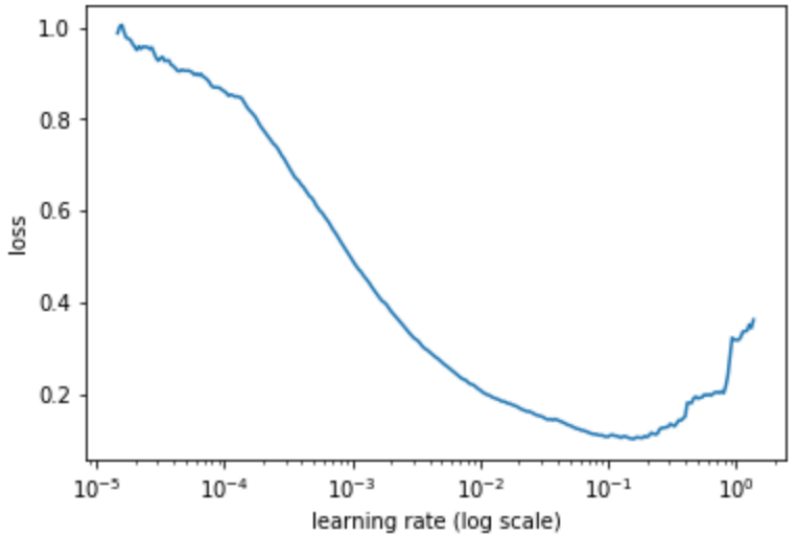
$$\tilde{\mathbf{x}} = \lambda \mathbf{x}_i + (1-\lambda)\mathbf{x}_j$$
$$\tilde{\mathbf{d}} = \lambda \mathbf{d}_i + (1-\lambda)\mathbf{d}_j$$

  - The weight λ is randomly chosen from [0,1]



Learning rate control

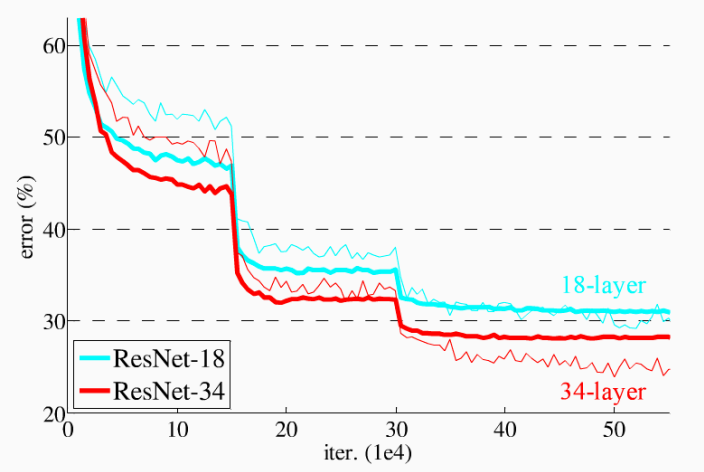
- The most important hyper-parameter: learning rate ϵ
  - It controls the step size of SGD
- Trial-and-error is inevitable for its selection
  - Depends on tasks and data
  - Grid search
    - Try many values systematically on the first few epochs



https://sgugger.github.io/how-do-you-find-a-good-learning-rate.html

Learning rate control

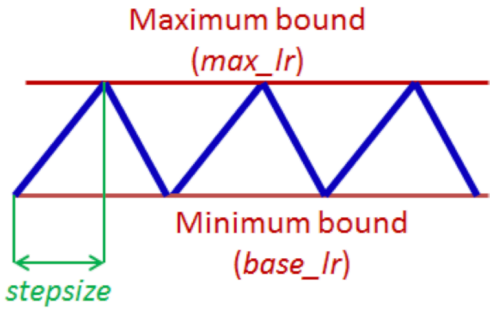
- Changing it during training often contributes to improved results
  - A standard practice: Use smaller values as training proceeds

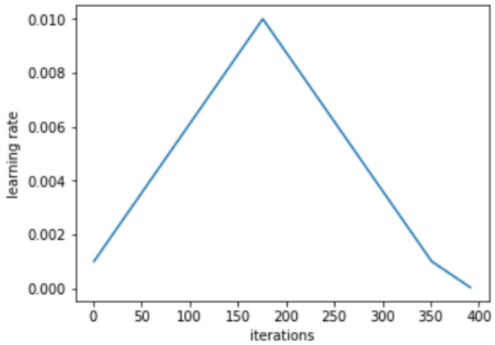


Starting from a value, scale it by 1/10 at appropriate timing
Or automatic schedule, e.g.

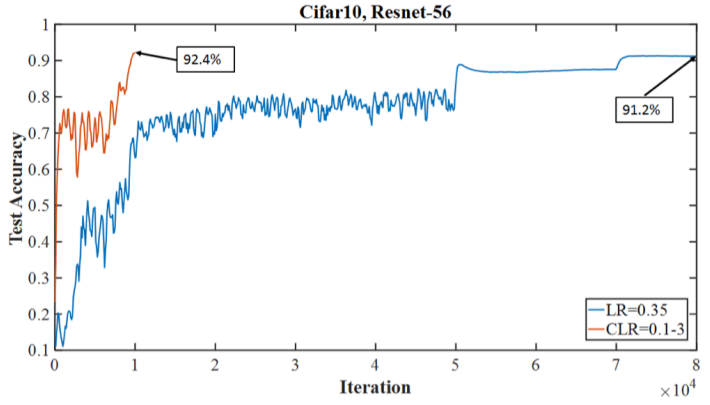
$$\epsilon_t = \epsilon_0 - \alpha t$$

  - Cyclical control
    - Cyclical learning rate; Smith (arXiv:1803.09820)





1 cycle method:
small→large→small→smallest



(a) An example of super-convergence.

AdaGrad

- Early method for automatically determining learning rate
  - Make update steps shorter for the elements which underwent large updates in the past

Normal SGD

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \epsilon \nabla E_t$$

rewriting

$$\Leftrightarrow \qquad \Delta w_{t,i} = -\epsilon g_{t,i}$$

$$\Delta \mathbf{w}_t (\equiv \mathbf{w}_{t+1} - \mathbf{w}_t) \qquad \mathbf{g}_t \equiv \nabla E_t$$

AdaGrad

$$\Delta w_{t,i} = -\frac{\epsilon}{\sqrt{\sum_{t'=1}^{t} g_{t',i}^2 + \varepsilon}} g_{t,i}$$

Sum over all previous updaets

To avoid division by zero

RMSProp/AdaDelta

- To resolve a drawback of AdaGrad: Step size eventually goes to zero
- Moving average over previous gradients is employed instead of their total sum

$$\langle g_i^2 \rangle_t = \gamma \langle g_i^2 \rangle_{t-1} + (1-\gamma) g_{t,i}^2$$

- RMSProp:

Root Mean Square

$$\Delta w_{t,i} = -\frac{\epsilon}{\sqrt{\langle g_i^2 \rangle_t + \varepsilon}} g_{t,i}$$

- AdaDelta replaces learning rate with moving average of updates

$$\Delta w_{t,i} = -\frac{\langle \Delta w_i \rangle_{t-1}}{\sqrt{\langle g_i^2 \rangle_t + \varepsilon}} g_{t,i}$$

  - where $\langle \Delta w_i^2 \rangle_t = \gamma \langle \Delta w_i^2 \rangle_{t-1} + (1-\gamma)(\Delta w_{t,i})^2$

Adam

Adaptive moment estimation

- Integrates RMSProp with momentum
  - Generalize the idea of adjusting updates with moving average of gradients
- Estimates of 1st and 2nd moments of gradients with their moving average:

$$m_{t,i} = \beta_1 m_{t-1,i} + (1-\beta_1) g_{t,i}$$
$$v_{t,i} = \beta_2 v_{t-1,i} + (1-\beta_2) g_{t,i}^2$$

← A variant: AdaMax
(l_2 norm replaced by l_∞)

- There are biases in them, which can be corrected as

$$\hat{m}_t = m_t / (1-\beta_1^t)$$
$$\hat{v}_t = v_t / (1-\beta_2^t)$$

$$v_t = (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2$$
$$\mathbb{E}[v_t] = \mathbb{E}\left[(1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} \cdot g_i^2\right]$$
$$= \mathbb{E}[g_t^2] \cdot (1-\beta_2) \sum_{i=1}^{t} \beta_2^{t-i} + \zeta$$
$$= \mathbb{E}[g_t^2] \cdot (1-\beta_2^t) + \zeta$$

- Adam:

$$\Delta w_{t,i} = -\frac{\epsilon}{\sqrt{\hat{v}_{t,i}} + \varepsilon} \hat{m}_{t,i}$$

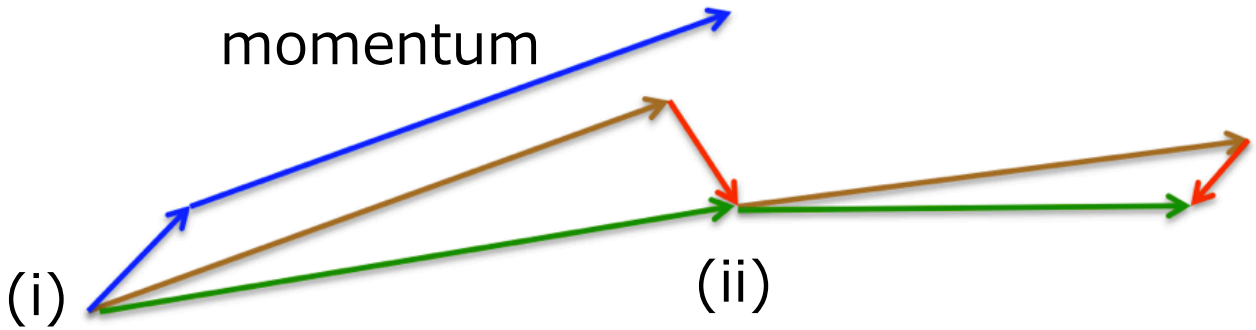
A variant: Nadam
(g_t is computed by Nesterov accelerated gradient)

Nesterov's accelerated gradient

- Gradient is computed not at current w but at w+momentum
  - Know to work well particularly for training RNNs

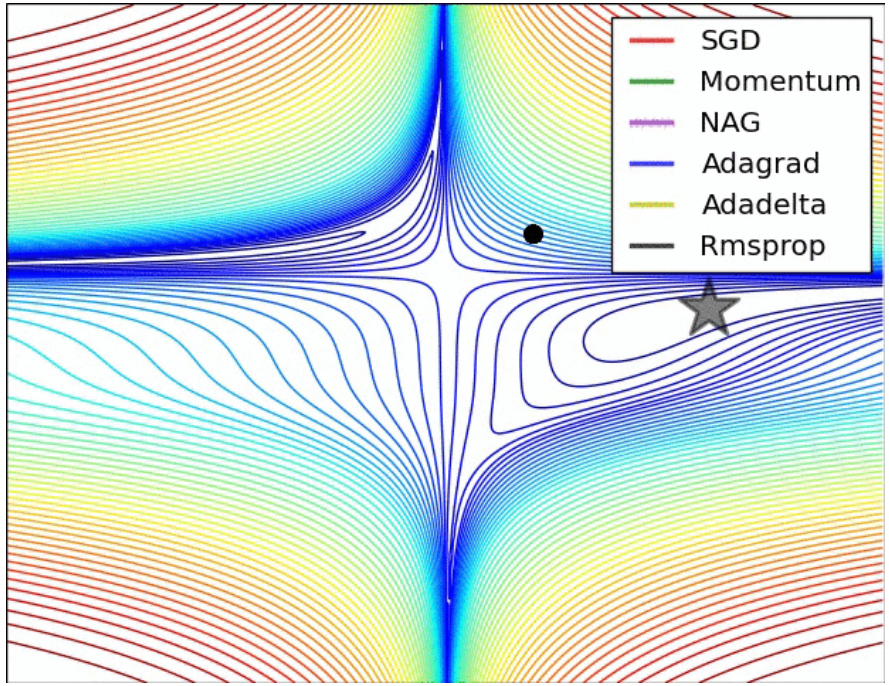
$$\nabla E_t\Big|_{\mathbf{w}=\mathbf{w}_t} \quad \Rightarrow \quad \nabla E_t\Big|_{\mathbf{w}=\mathbf{w}_t+\mu\mathbf{v}_t}$$

(i) (ii)

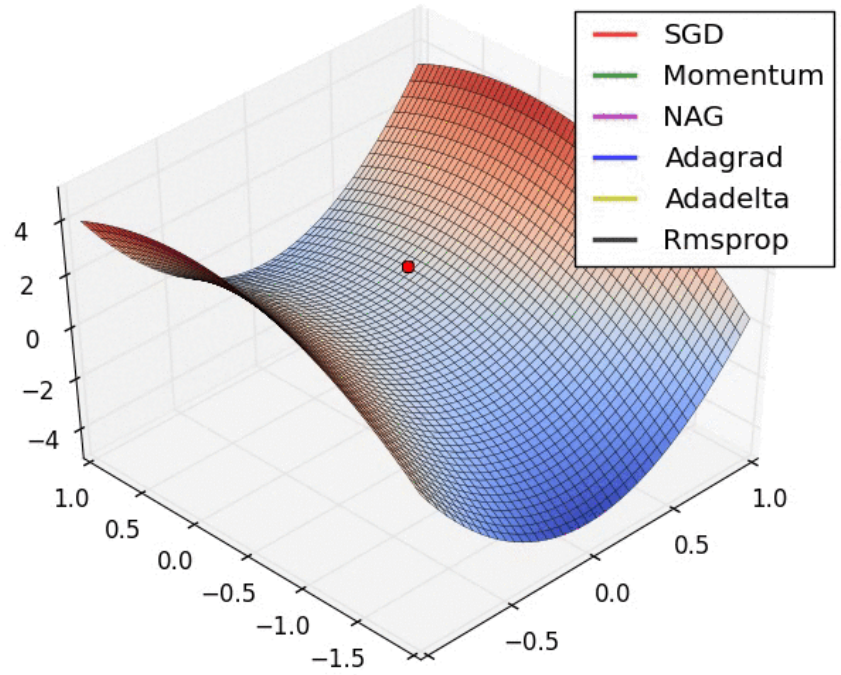


Behavior of various optimizers

- In this example
  - Momentum overshoot; NAG works better
  - Adagrad/Adadelta/RMSprop work equally well
  - SGD is slow



- Behavior around a saddle point
  - AdaDelta is the fastest
  - Adagrad/RMSprop perform similarly
  - Momentum/NAG are slow
  - SGD is trapped forever



Images credit: Alec Radford.