CS60010: Deep Learning

Sudeshna Sarkar

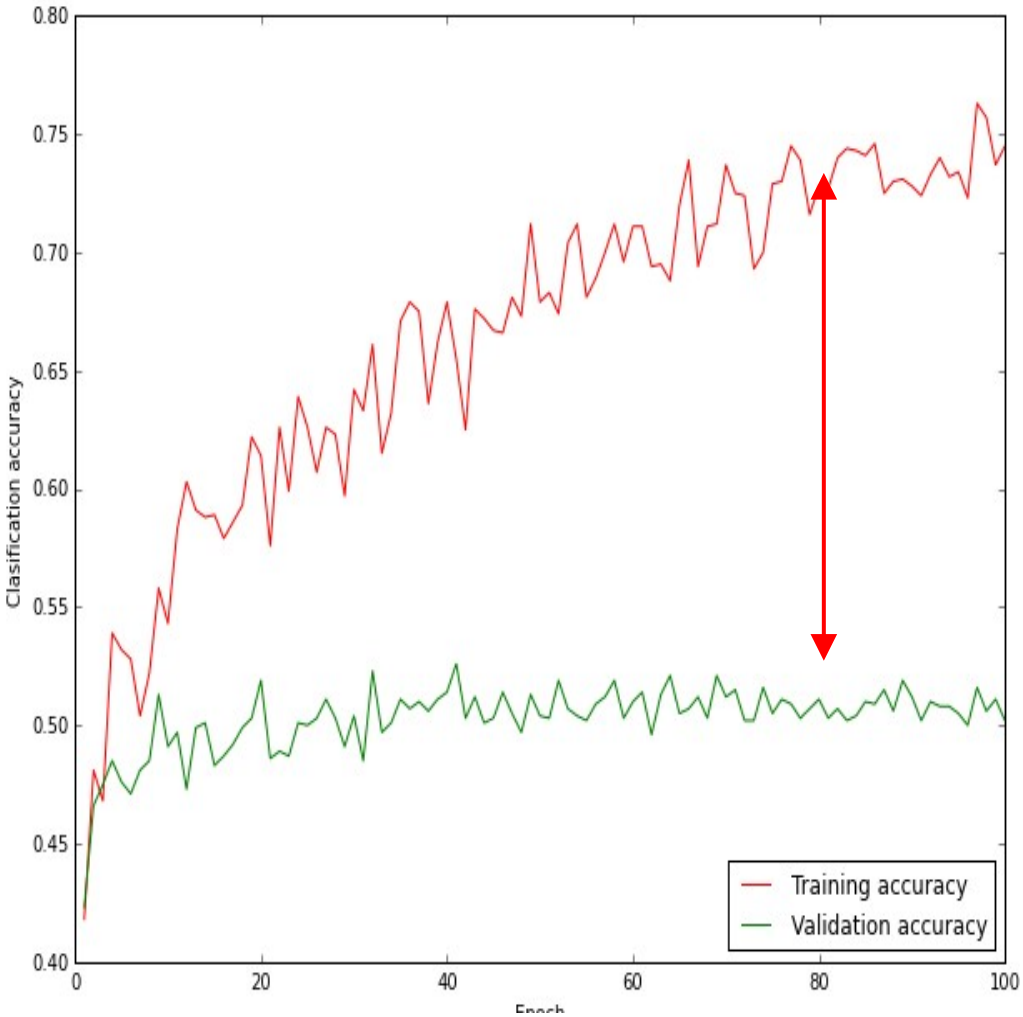
Spring 2018

23 Jan 2018

Training

1. Check that loss is reasonable.
   - Loss goes up as you increase regularization.
2. Make sure that you can overfit very small portion of the training data
3. Start with small regularization and find learning rate that makes the loss go down.
   - loss not going down: means learning rate too low
   - loss exploding: learning rate too high

Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Cross-validation strategy

May do **coarse -> fine** cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Track the ratio of weight updates / weight magnitudes

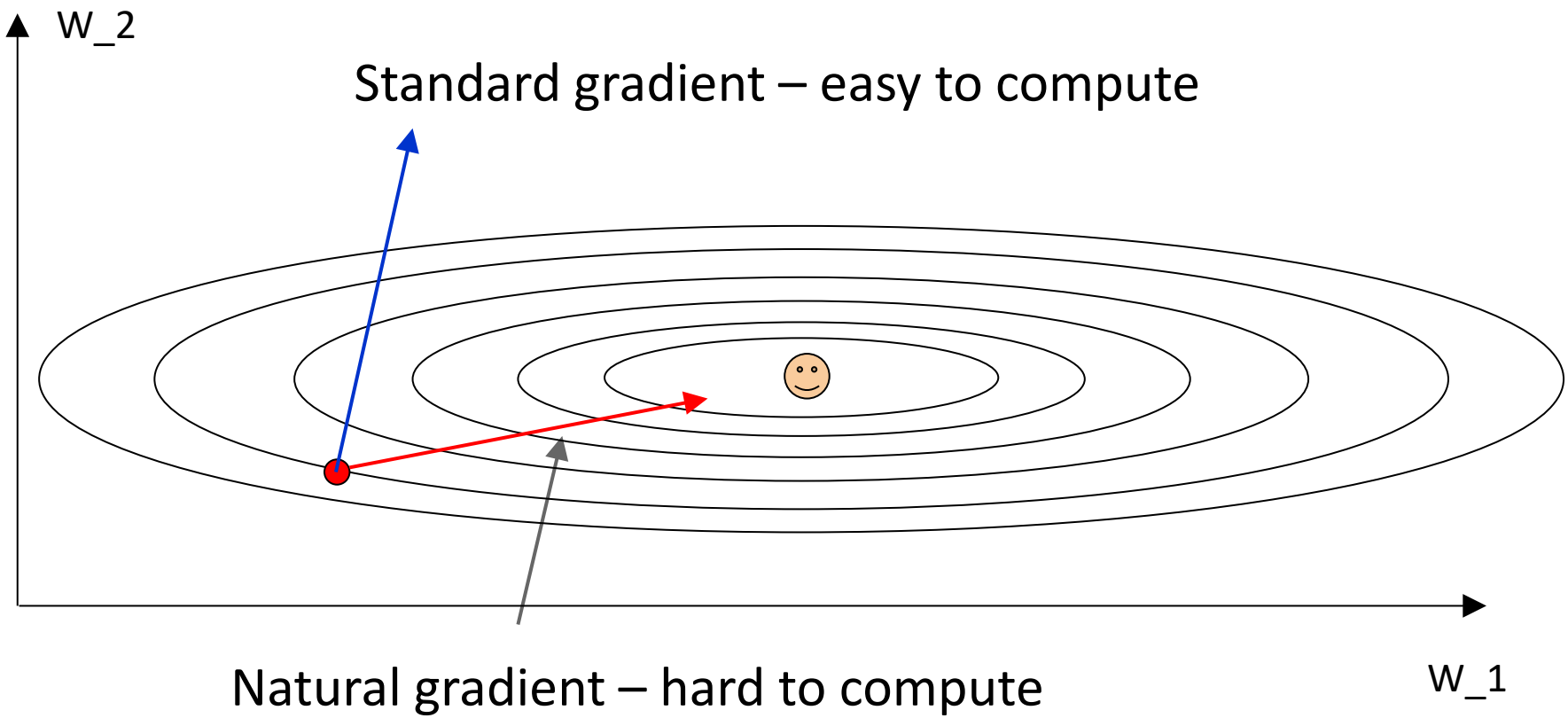
```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

ratio between the values and updates: ~ 0.0002 / 0.02 = 0.01 (about okay)
want this to be somewhere around 0.001 or so

Activation / Gradient distributions per layer

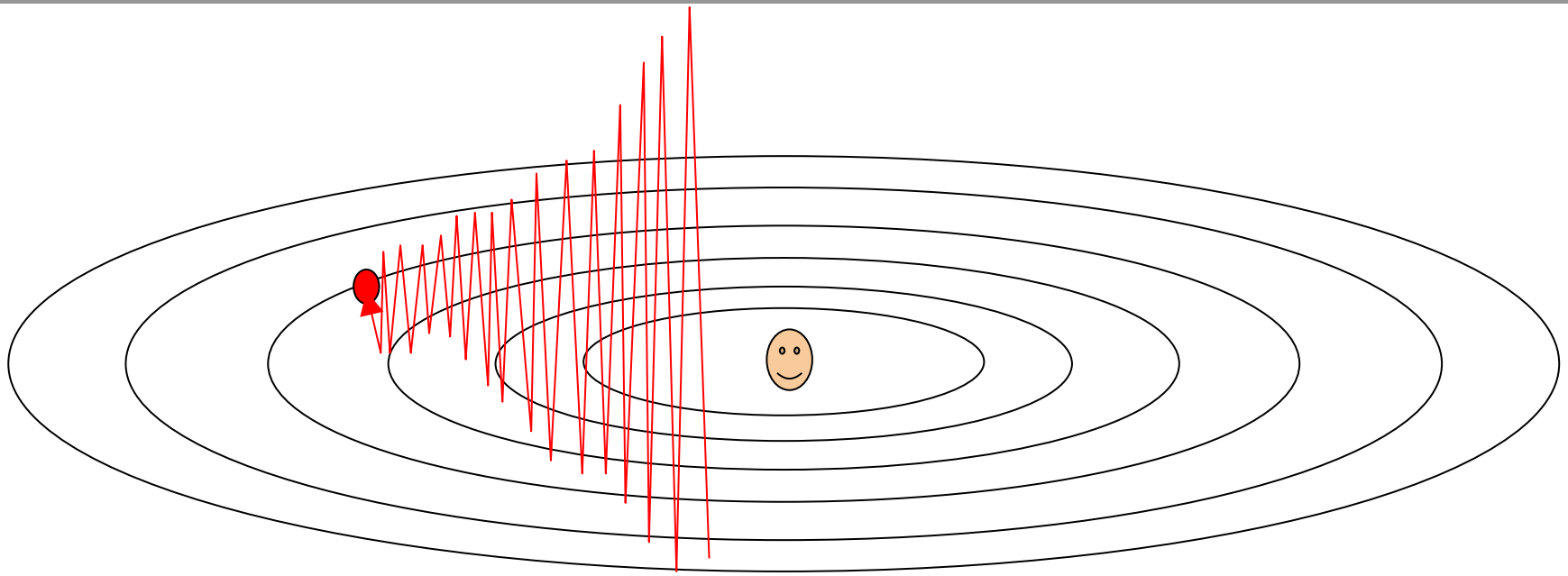
- An incorrect initialization can slow down or even completely stall the learning process
- Plot activation/gradient histograms for all layers of the network.

Back to gradients



Natural gradient : Rather than treating a change in every parameter equally, we need to scale each parameter's change according to how much it affects our network's entire output distribution.

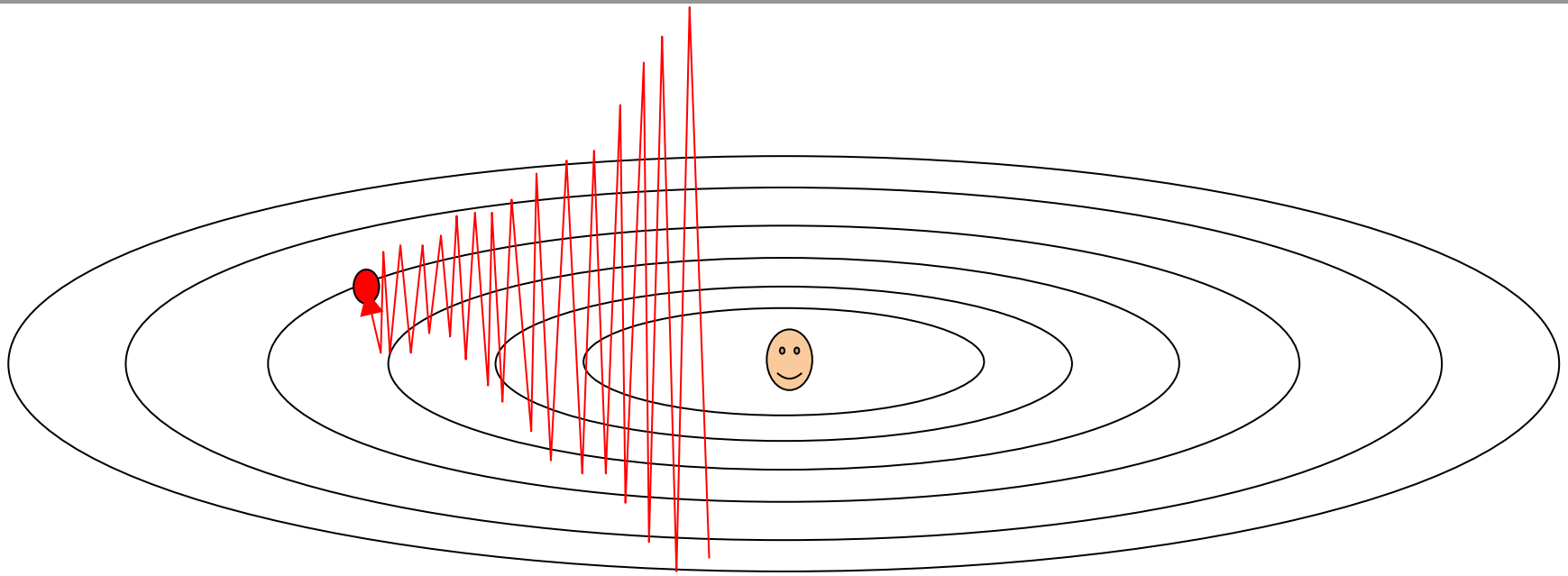
Gradient Magnitudes:



Gradients too big → divergence

Gradients too small → slow convergence

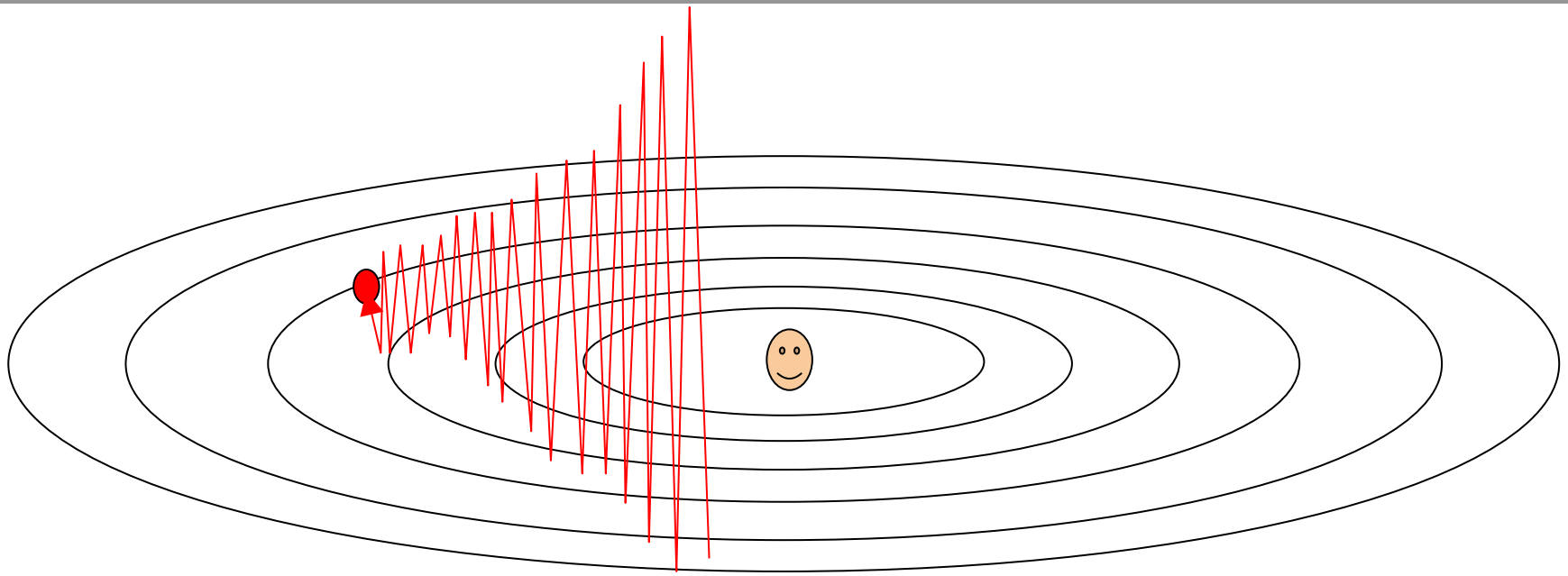
Gradient Magnitudes:



Gradients too big → divergence
Gradients too small → slow convergence

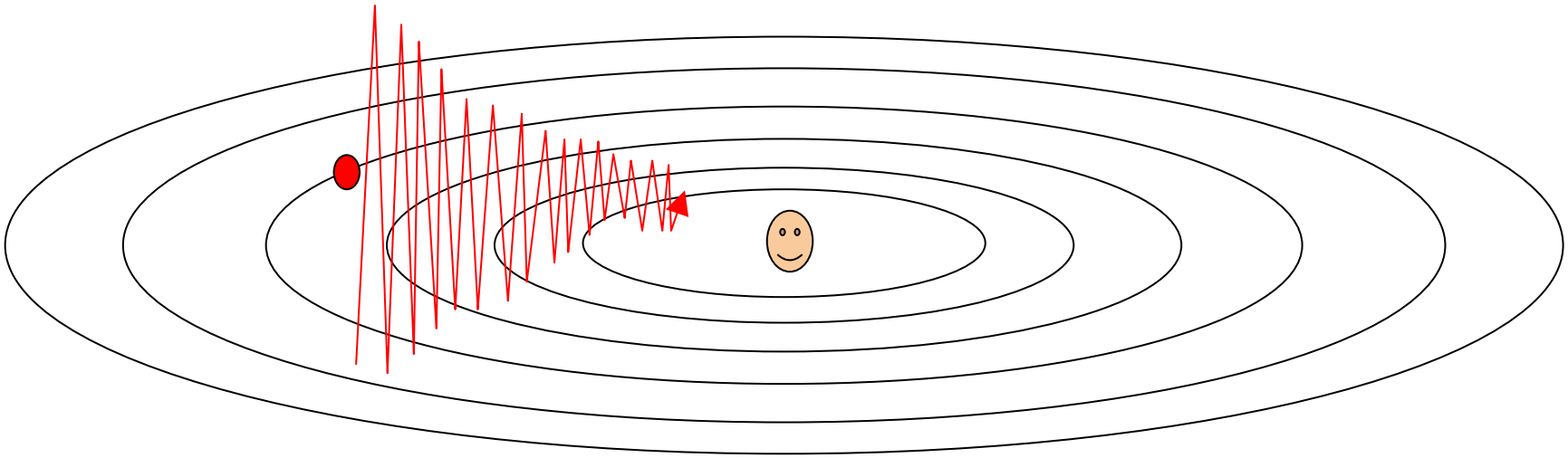
Divergence is much worse!

Gradient Magnitudes



What's the simplest way to ensure gradients stay bounded?

Gradient clipping

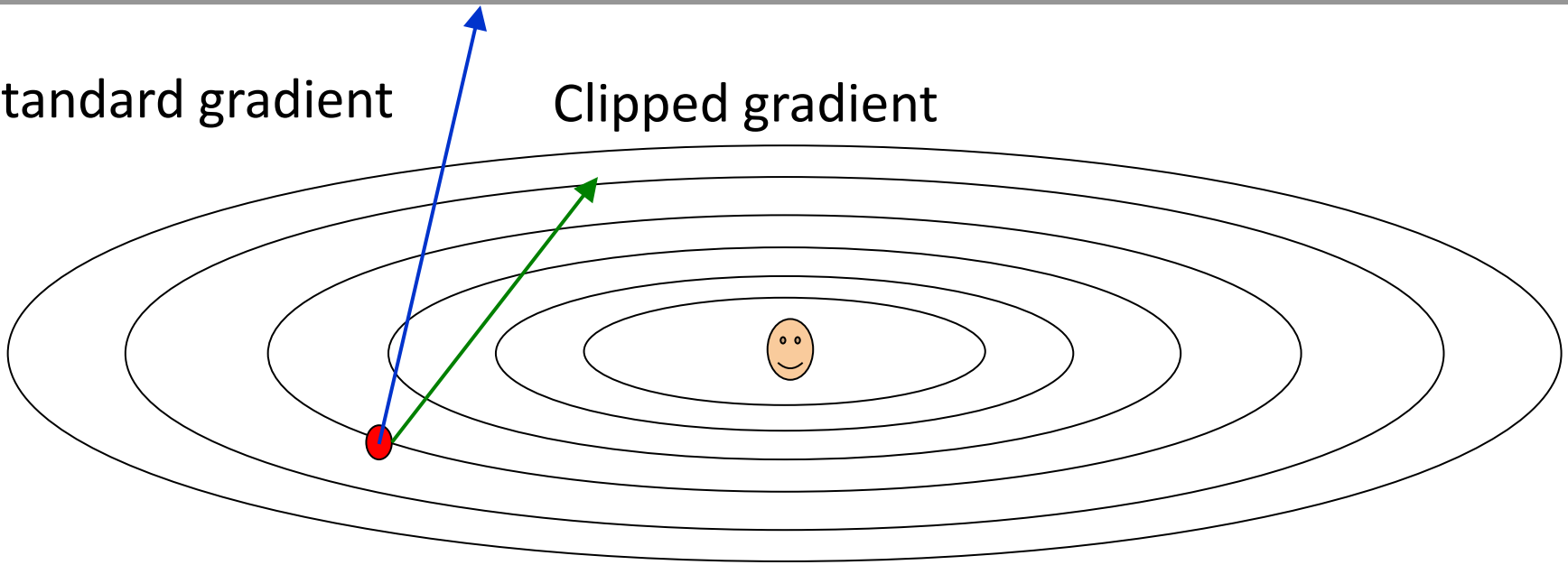


Simply limit the magnitude of each gradient:

$$\overline{g_i} = \min(g_{\max}, \max(-g_{\max}, g_i))$$

so $|\overline{g_i}| \leq g_{\max}$. Then use a decreasing learning rate to converge to an optimum.

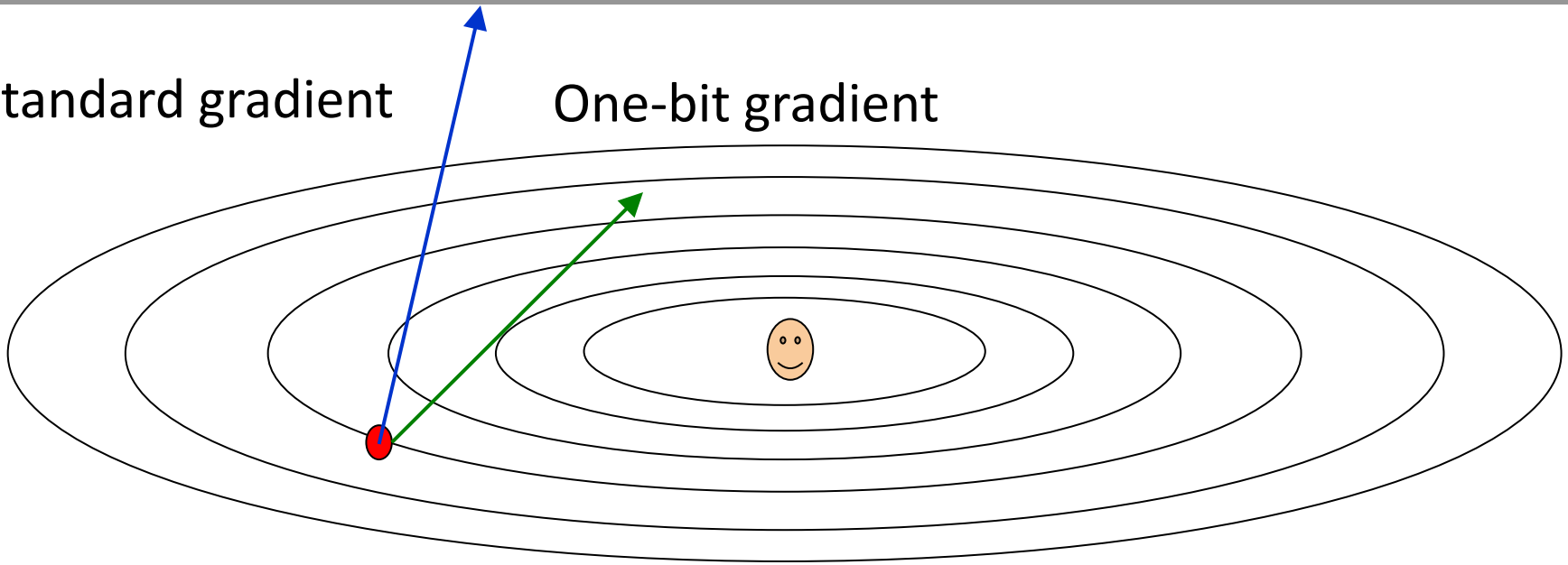
Extreme Gradient clipping



Gradient clipping limits the largest gradient dimensions, while others may be very small.
ADAGRAD and RMSprop scale gradient dimensions by the inverse std deviation, so all dimension have unit sdev.

What if we scale gradients up before clipping, so all dimensions are clipped?

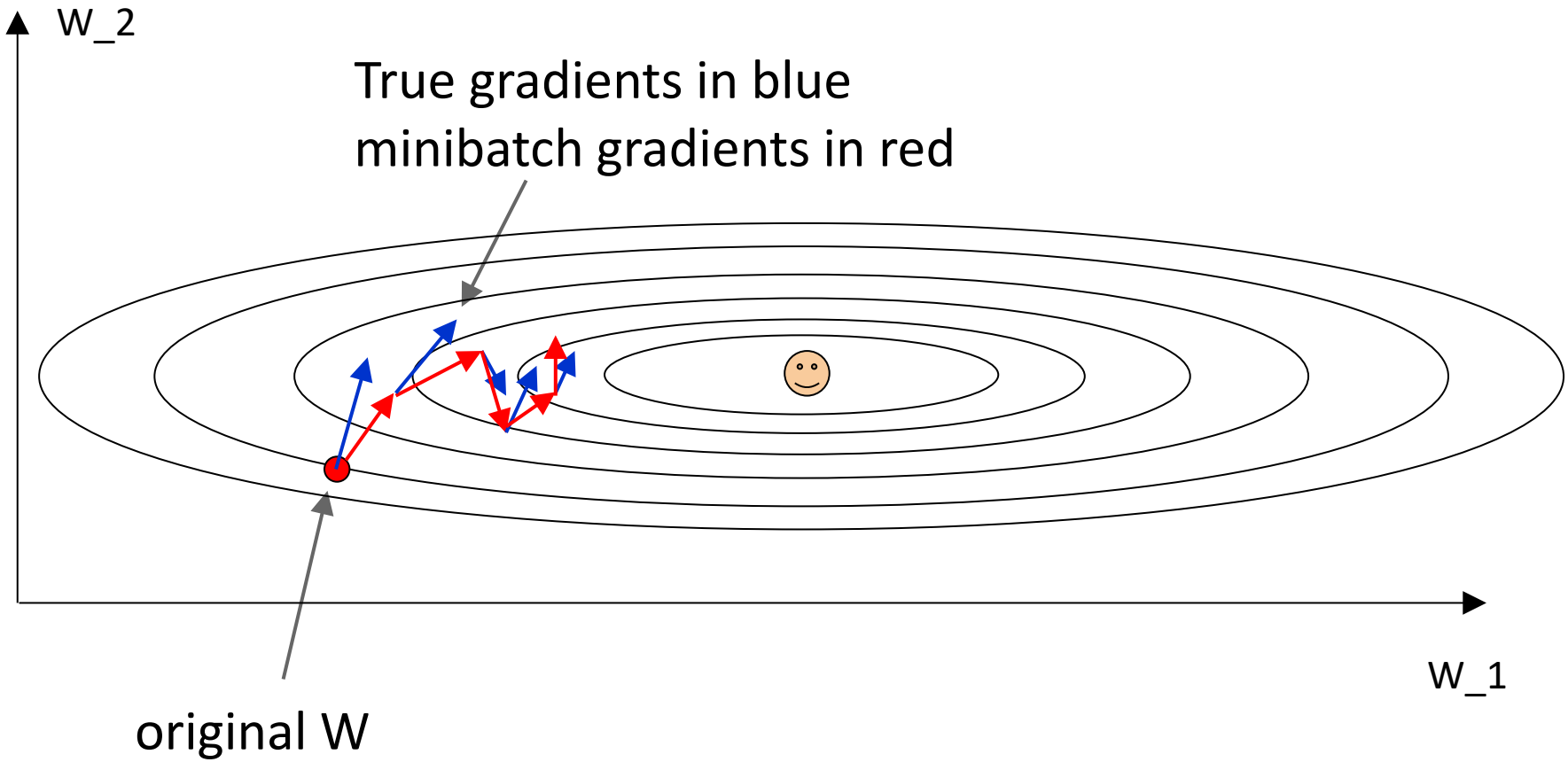
One-bit gradients



If we clip all gradient dimensions, we are left only with their sign: $\bar{g}_i = g_{\max}(-1,1,1,-1,1,\dots)$

This actually works on some problems with little or no loss of accuracy: (see "1-Bit Stochastic Gradient Descent and Application to Data-Parallel Distributed Training of Speech DNNs" by Seide et al. 2014)

Stochastic Gradient Descent



Gradients are noisy but still make good progress on average

Gradient Noise

If a little noise is good, what about **adding** noise to gradients?

A: Works Great for many models!

Is especially valuable for complex models that would overfit otherwise.

"Adding Gradient Noise Improves Learning for Very Deep Networks" Arvind Neelakantan et al., 2016

Gradient Noise

Schedule:

$$g_t \leftarrow g_t + N(0, \sigma_t^2)$$

where the noise variance is:

$$\sigma_t^2 = \frac{\eta}{(1+t)^\gamma}$$

with η selected from $\{0.01, 0.3, 1.0\}$ and $\gamma = 0.55$.

Gradient Noise

Results on MNIST with a 20-layer ReLU network:

Experiment 1: Simple Init, No Gradient Clipping

Setting	Best Test Accuracy	Average Test Accuracy
No Noise	89.9%	43.1%
With Noise	96.7%	52.7%
No Noise + Dropout	11.3%	10.8%

Experiment 2: Simple Init, Gradient Clipping Threshold = 100

No Noise	90.0%	46.3%
With Noise	96.7%	52.3%

Experiment 3: Simple Init, Gradient Clipping Threshold = 10

No Noise	95.7%	51.6%
With Noise	97.0%	53.6%

Experiment 4: Good Init (Sussillo & Abbott, 2014) + Gradient Clipping Threshold = 10

No Noise	97.4%	92.1%
With Noise	97.5%	92.2%

Experiment 5: Good Init (He et al., 2015) + Gradient Clipping Threshold = 10

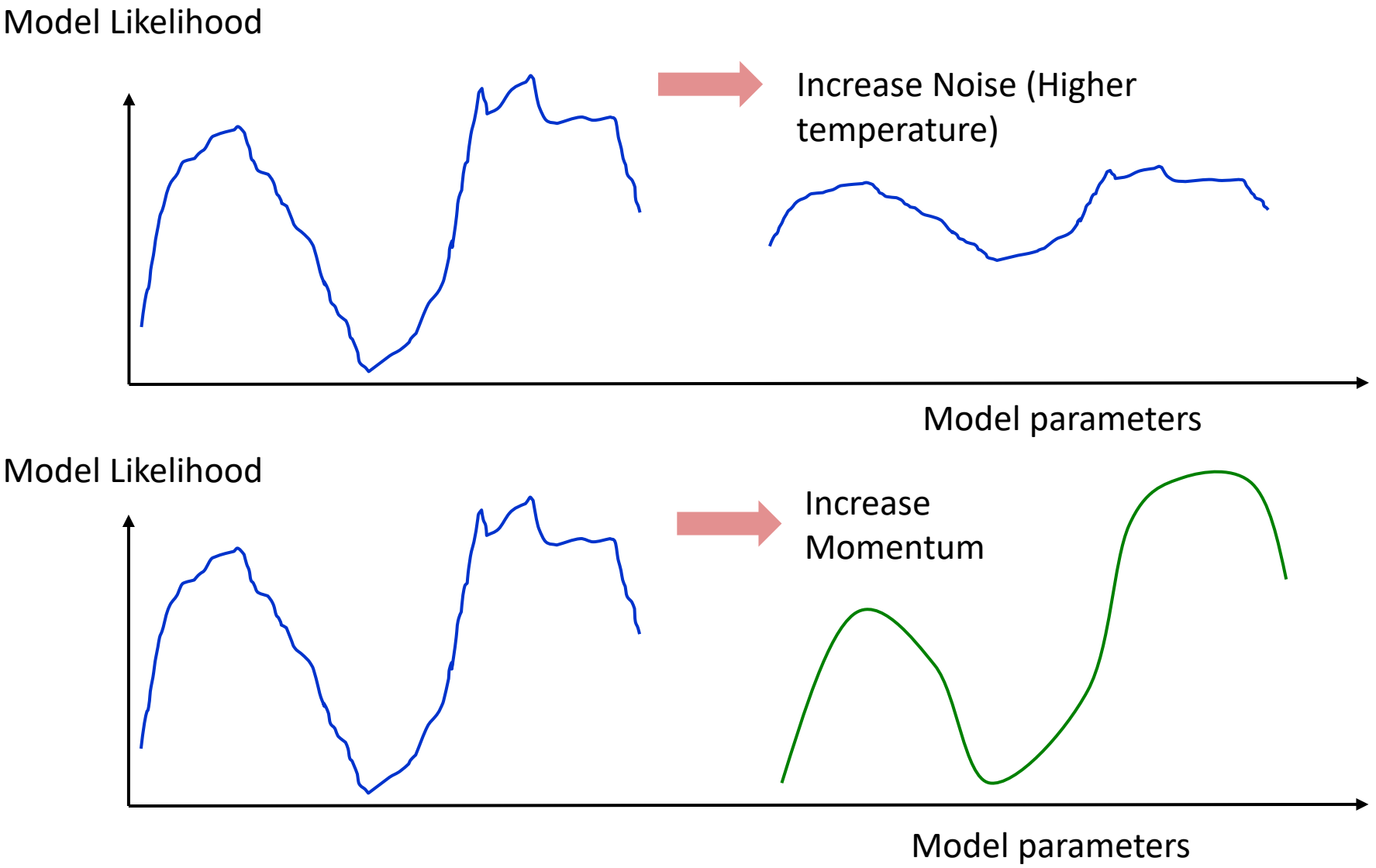
No Noise	97.4%	91.7%
With Noise	97.2%	91.7%

Experiment 6: Bad Init (Zero Init) + Gradient Clipping Threshold = 10

No Noise	11.4%	10.1%
With Noise	94.5%	49.7%

Table 1: Average and best test accuracy percentages on MNIST over 40 runs. Higher values are better.

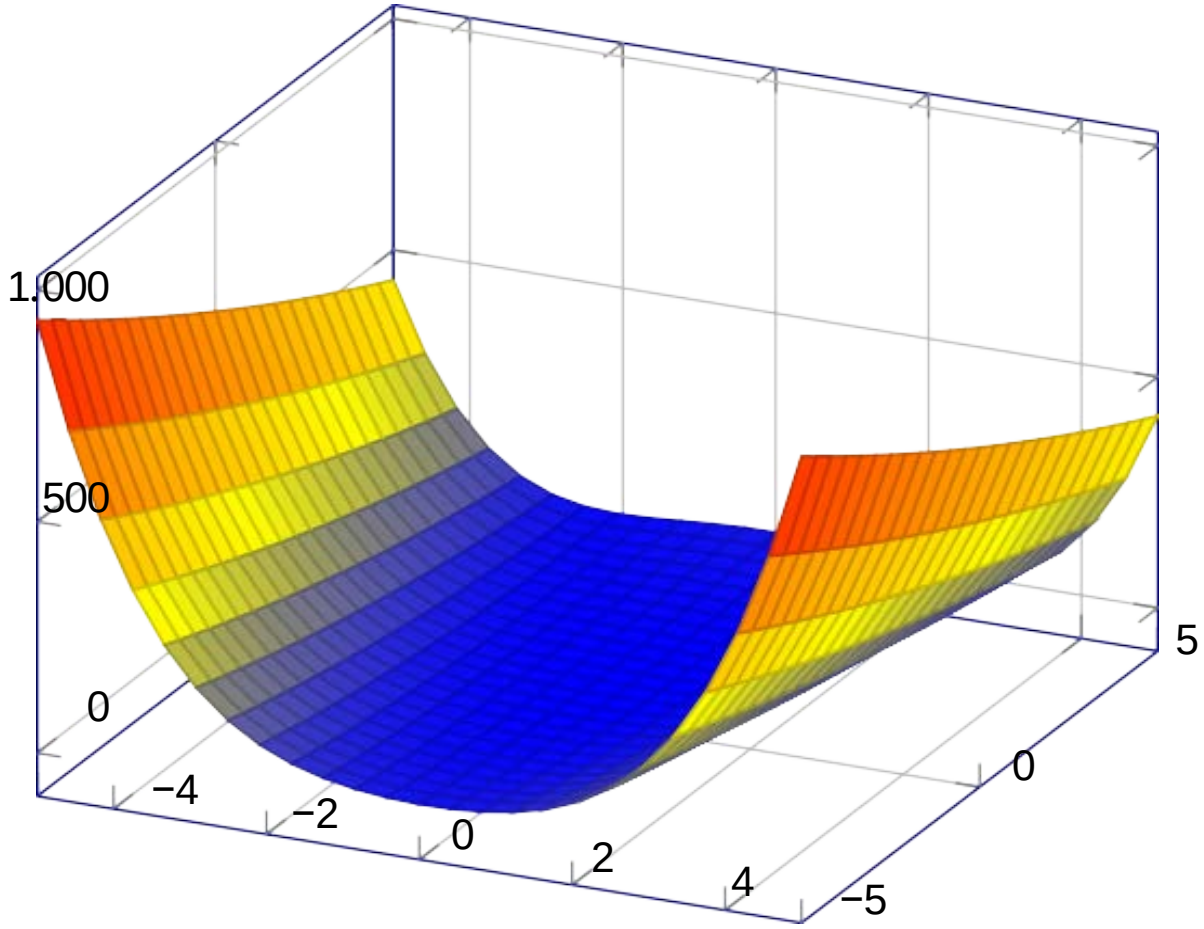
Gradient Noise + Momentum



Momentum

- The Momentum method is a method to accelerate learning using SGD
- In particular SGD suffers in the following scenarios:
  - Error surface has high curvature
  - We get small but consistent gradients
  - The gradients are very noisy

Momentum



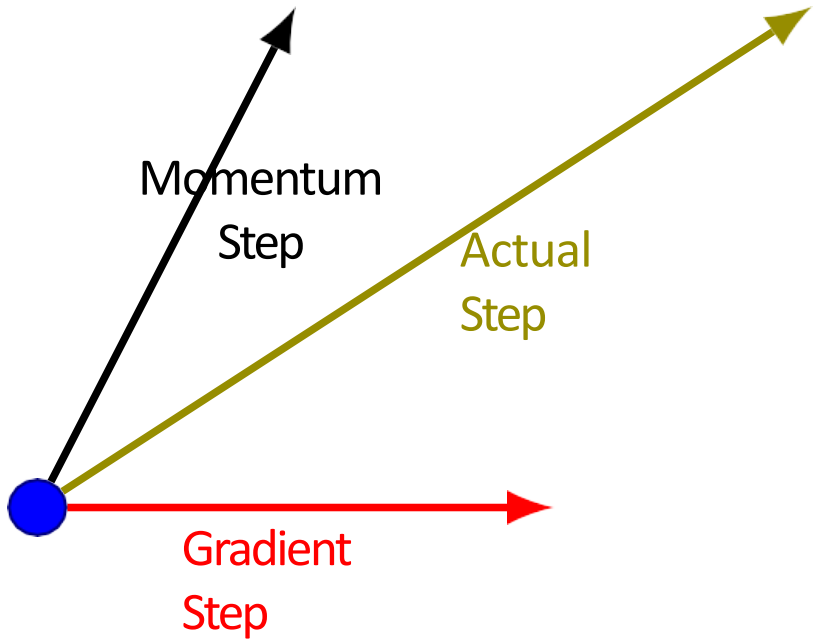
Gradient Descent would move quickly down the walls, but very slowly through the valley floor

Momentum

- Introduce a new variable v, the velocity
  - We think of v as the direction and speed by which the parameters move as the learning dynamics progresses
  - The velocity is an exponentially decaying moving average of the negative gradients

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left(L\left(f\left(x^{(i)}; \theta \right), y^{(i)} \right) \right)$$

- Update rule: $\theta \leftarrow \theta + v$



  - With Momentum update, the parameter vector will build up velocity in any direction that has consistent gradient.
  - The velocity accumulates the previous gradients.

Momentum

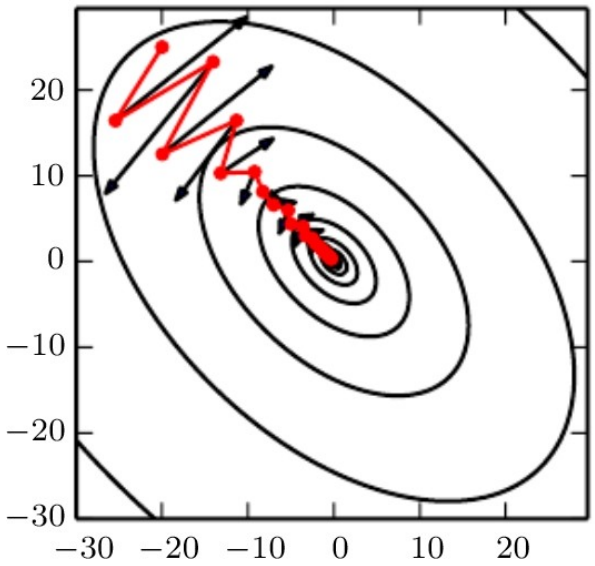
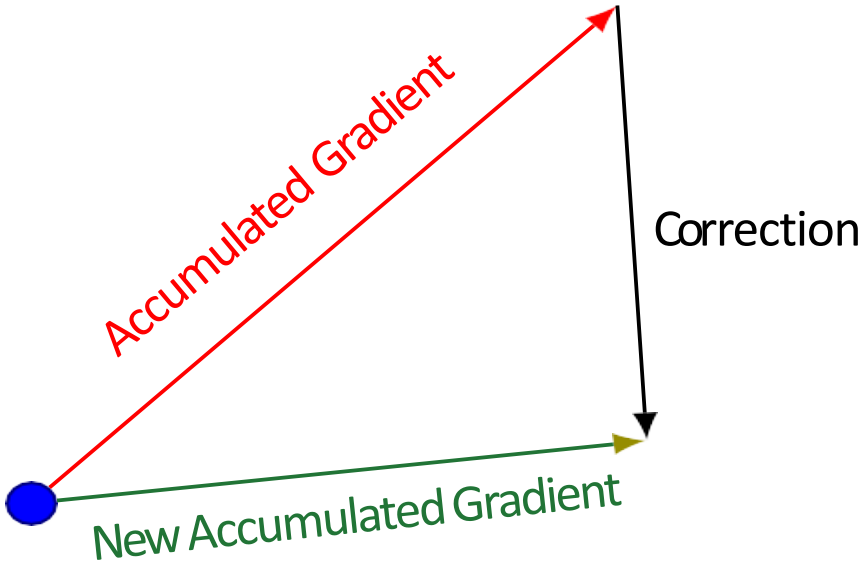


Illustration of how momentum traverses such an error surface better compared to Gradient Descent

Nesterov Momentum

First take a step in the direction of the accumulated gradient

Then calculate the gradient and make a correction

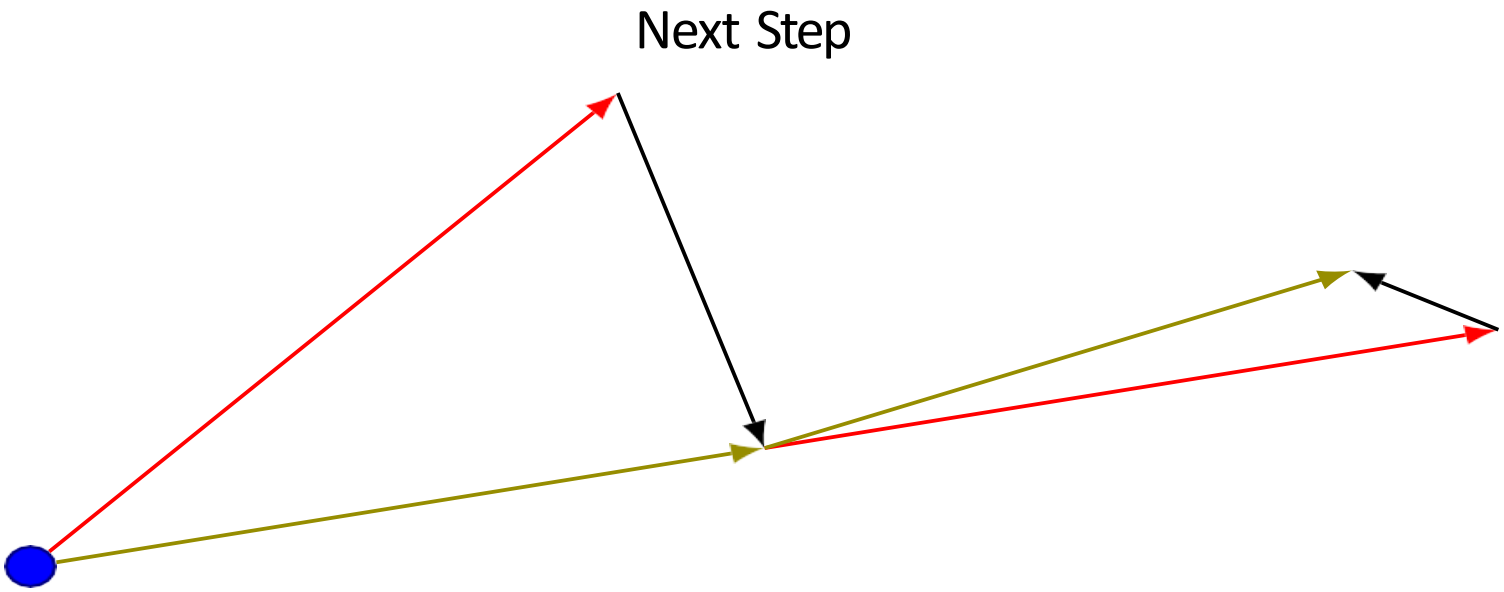


Momentum: $v_t \leftarrow \alpha v_{t-1} - \epsilon \nabla_\theta \left(L\left(f\left(x^{(i)}; \theta \right), y^{(i)} \right) \right)$

Nesterov: $v_t \leftarrow \alpha v_{t-1} - \epsilon \nabla_\theta \left(L\left(f\left(x^{(i)}; \theta + \alpha v_{t-1} \right), y^{(i)} \right) \right)$

$$\theta \leftarrow \theta + v$$

Nesterov Momentum



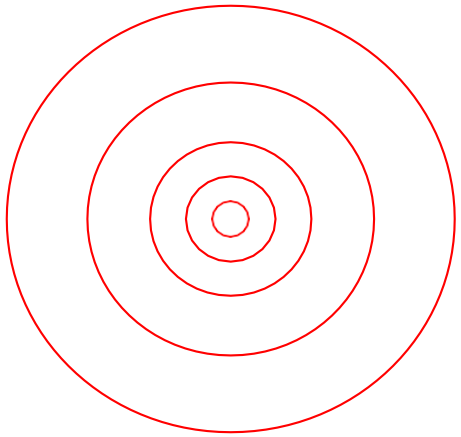
Momentum: $v_t \leftarrow \alpha v_{t-1} - \epsilon \nabla_\theta \left(L\left(f\left(x^{(i)}; \theta \right), y^{(i)} \right) \right)$

Nesterov: $v_t \leftarrow \alpha v_{t-1} - \epsilon \nabla_\theta \left(L\left(f\left(x^{(i)}; \theta + \alpha v_{t-1} \right), y^{(i)} \right) \right)$

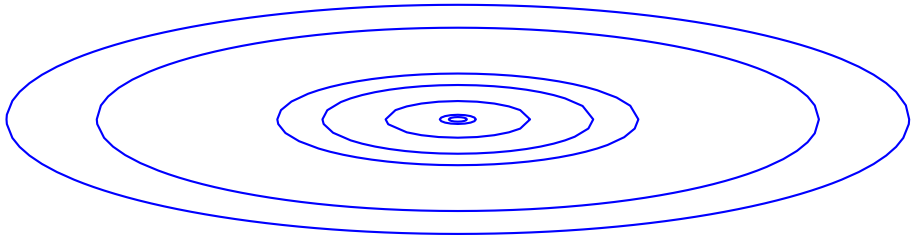
$$\theta \leftarrow \theta + v$$

Adaptive Learning Rate Methods

- Till now we assign the same learning rate to all features
- If the features vary in importance and frequency, this may not be a good idea



Nice (all features are equally important)



Harder!

AdaGrad

- Many features are irrelevant, rare features are often informative.
- Adagrad provides a feature-specific adaptive learning rate by

- 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

RMSProp

- AdaGrad is good when the objective is convex.
- AdaGrad might shrink the learning rate too aggressively, we want to keep the history in mind
- We can adapt it to perform better in non-convex settings by accumulating an exponentially decaying average of the gradient

Adam

Adam is like RMSProp with Momentum but with bias correction terms for the first and second moments

Adam update rule consists of the following steps

- Compute gradient g_t at current time t
- Update biased first moment estimate

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

- Update biased second raw moment estimate

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

- Compute bias-corrected first moment estimate

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

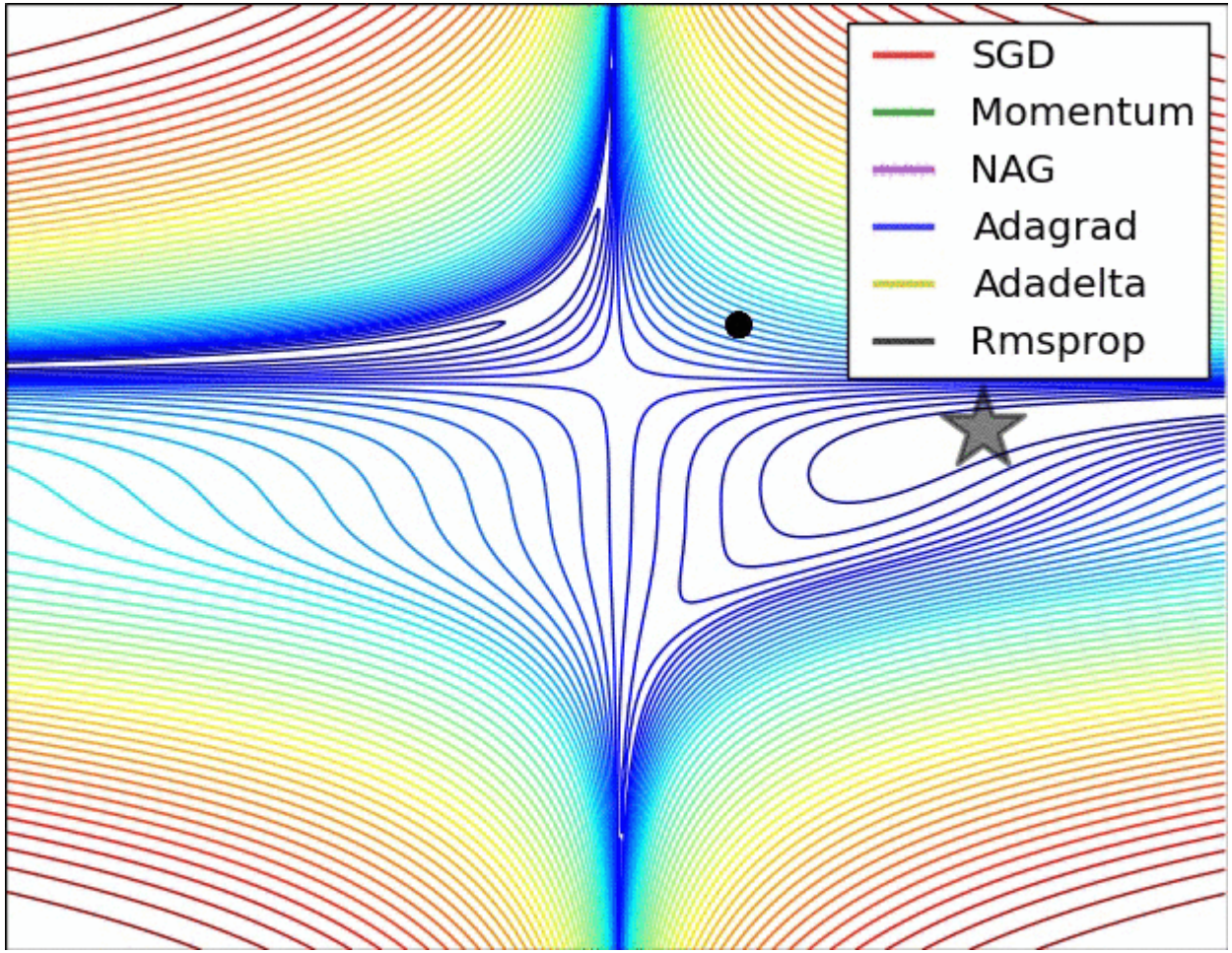
- Compute bias-corrected second raw moment estimate

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

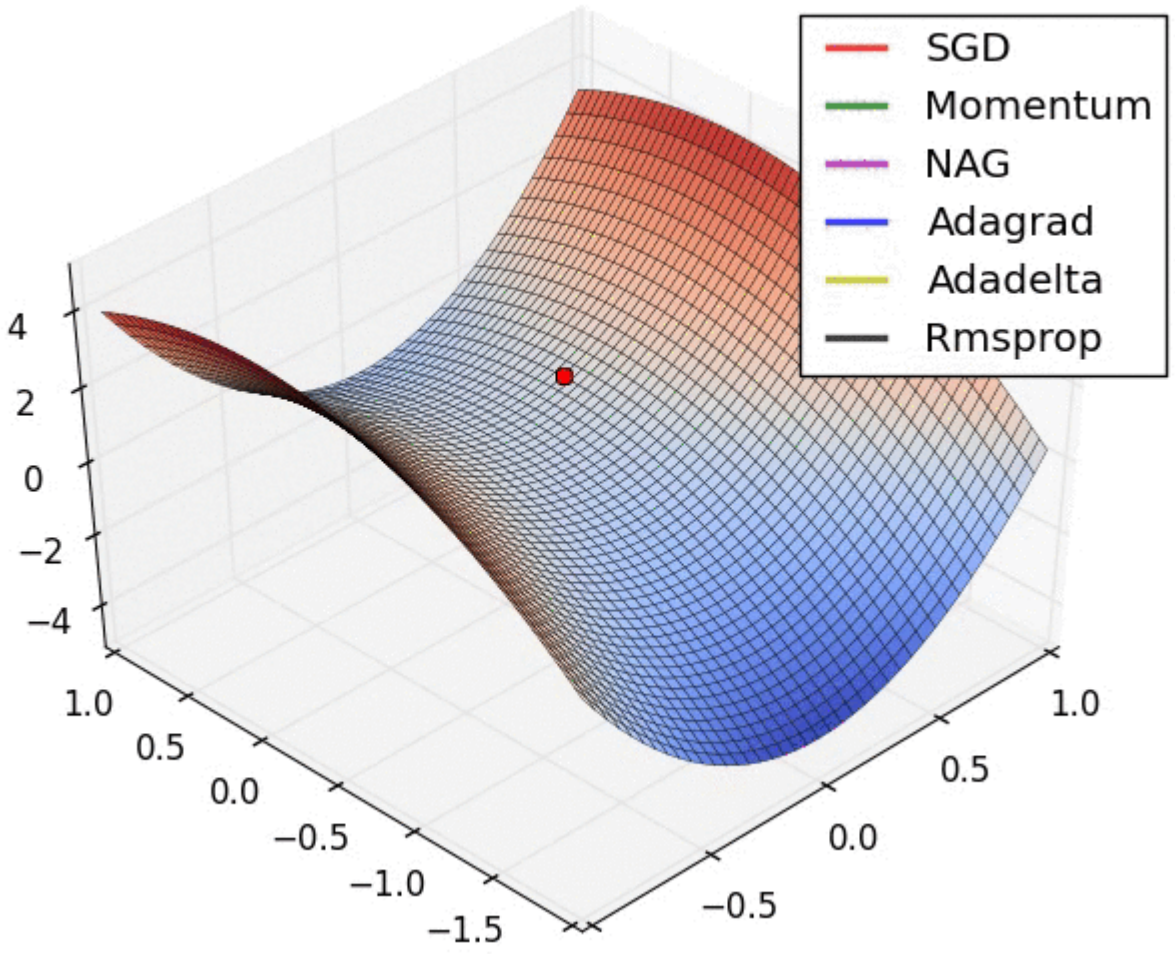
- Update parameters

$$\theta_t = \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Visualization: SGD optimization on loss surface contours



Visualization: SGD optimization on saddle point



Visualizing and Animating Optimization Algorithms with Matplotlib

- http://louistiao.me/notes/visualizing-and-animating-optimization-algorithms-with-matplotlib/