# CS60010: Deep Learning

## Spring 2023

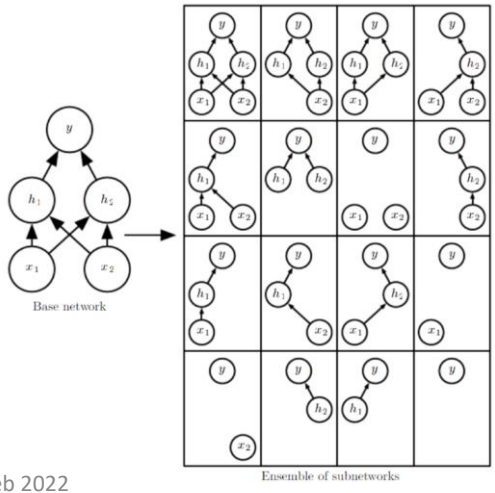
Sudeshna Sarkar

**Regularization and Optimization- Part 2**

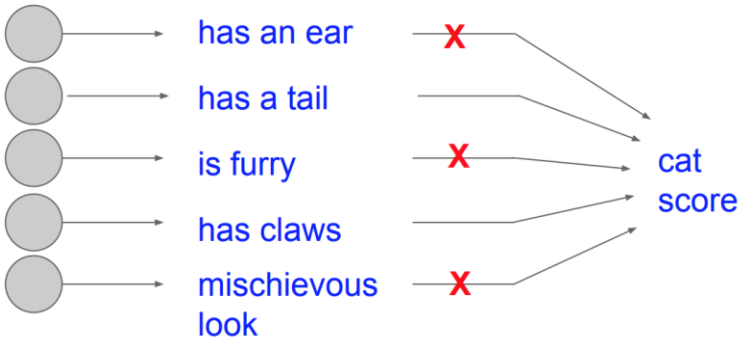
2 Feb 2023

# Regularization Strategies: Dropout

- Bagging is a technique for reducing generalization error through combining several models (Breiman, 1994)
- Bagging: (1) Train k different models on k different subsets of training data, constructed to have the same number of examples as the original dataset through random sampling from that dataset with replacement
- Bagging: (2) Have all of the models vote on the output for test examples
- Dropout is a computationally inexpensive but powerful extension of Bagging
- Training with dropout consists of training sub-networks that can be formed by removing non-output units from an underlying base network




Forces the network to have a redundant representation.

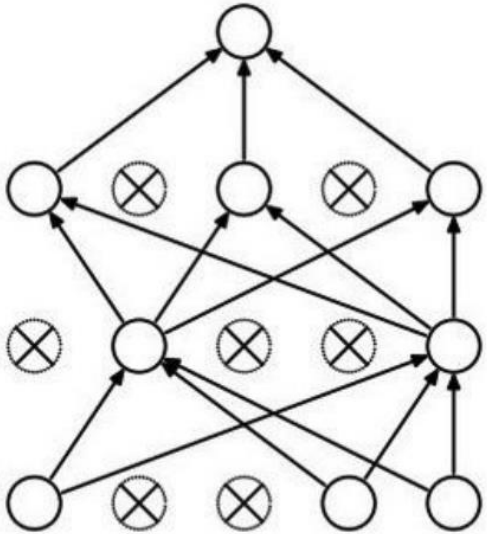





Images courtesy: Goodfellow et. al., Karpathy et. al.

# Dropout – At Test Time

- Ideally, the randomness would have to be integrated out.
- Monte Carlo approximation: Do many forward passes with different random neurons dropped out. Then average out all predictions.
- An approximation to this approximation:
  - Can this be done in a single forward pass!
  - Can this be done without dropping out any neuron during forward pass at test time!
  - 1st way: Get the output of the network at test time with all neurons on. Scale down this by multiplying it with the probability value with which neurons are dropped during training.
  - 2nd way: During training compute the output of the network that you get after dropping out neurons with probability ‘p’. During training itself, scale up this by multiplying it with (1/p). At test time, get the output as what is coming by keeping all the neurons on.



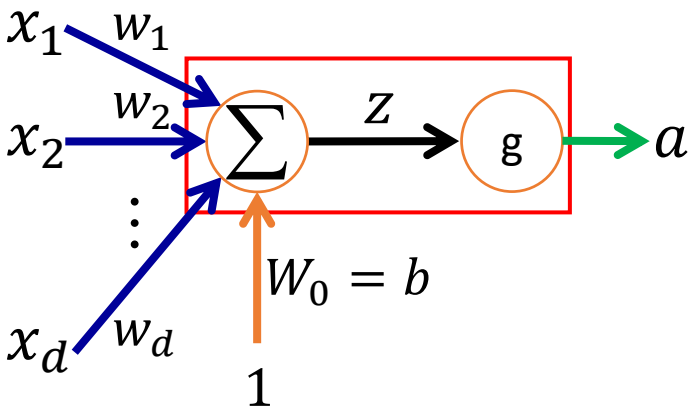
# Batch Normalization (batchnorm)

- Training deep neural networks can be sensitive to the initial random weights and configuration of the learning algorithm.
- The distribution of the inputs to layers deep in the network may change after each mini-batch when the weights are updated.
- *Batch normalization is a technique for training very deep neural networks that standardizes the inputs to a layer for each mini-batch*. This has the effect of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

# Batch Normalization

- BN mitigates the interdependency between layers during training.
- BN makes the optimisation landscape smoother

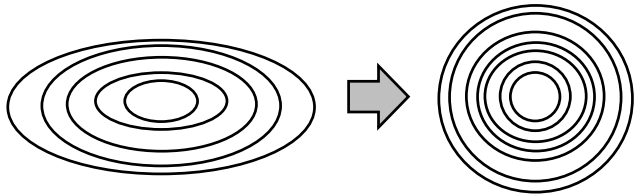
# Batch Normalization

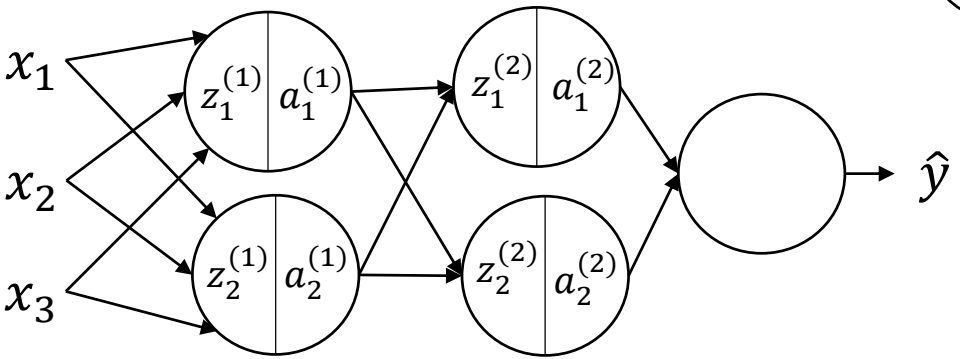



$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$$

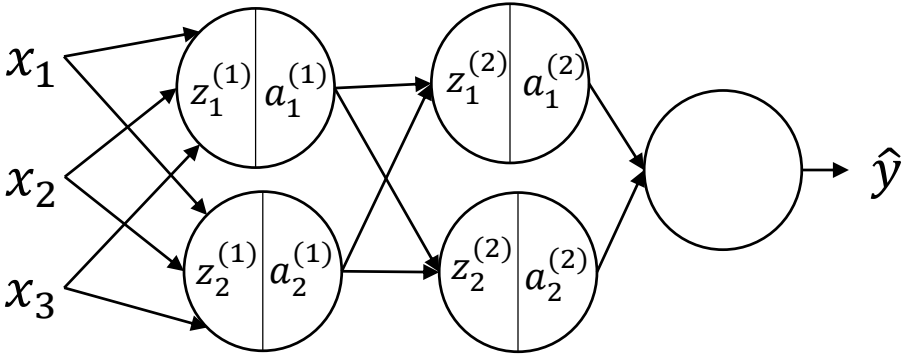
$$X = \mathrm{X} - \mu$$

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)2}$ (elementwise)





# Batch Normalization:1. Normalize net inputs




$$\mu_j = \frac{1}{m}\sum_{i=1}^{m} z_j^{(i)}$$

$$\sigma_j^2 = \frac{1}{m}\sum_{i=1}^{m}\left(z_j^{(i)} - \mu_j\right)^2$$

$$z'^{[i]}_j = \frac{z_j^{(i)} - \mu_j}{\sigma_j}$$

In practice,

$$z'^{[i]}_j = \frac{z_j^{(i)} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

# BatchNorm Step 2: Pre-Activation Scaling

$$z'^{[i]}_j = \frac{z^{(i)}_j - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$$

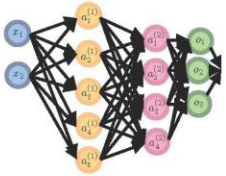
$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

Controls the spread

Controls the mean

Learnable parameters

# BatchNorm Step 1 & 2 Summarized



$$z'^{[i]}_j = \frac{z^{[i]}_j - \mu_j}{\sigma_j}$$

$$a'^{[i]}_j = \gamma_j \cdot z'^{[i]}_j + \beta_j$$

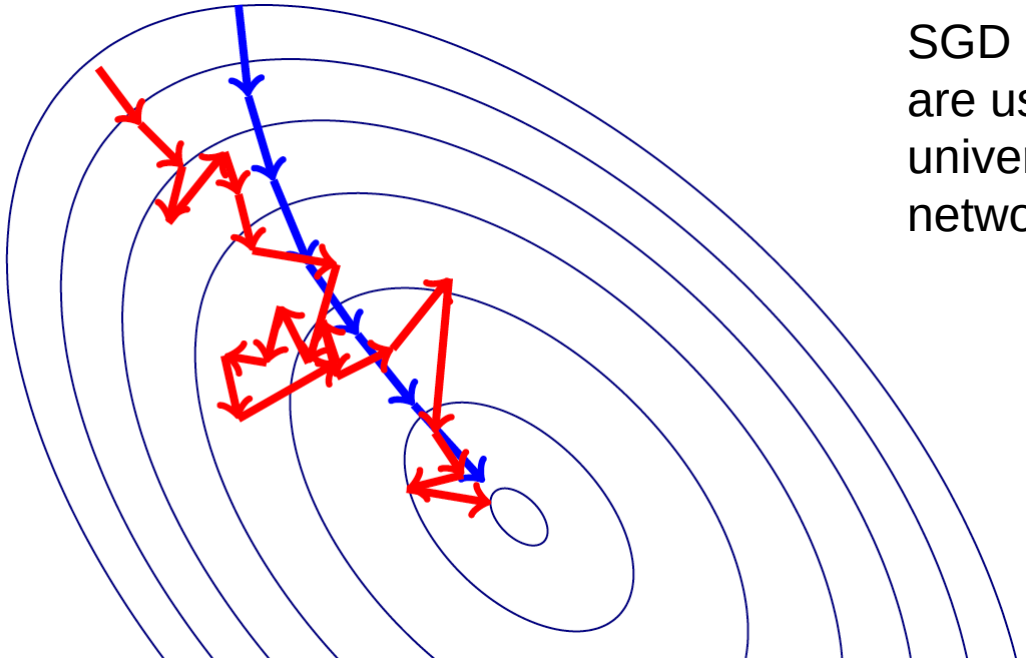
$x_1$ → $z^{(1)}_1$ → $z'^{(1)}_1$ → $a'^{(1)}_1$ → $a^{(1)}_1$ first hidden layer

$z^{(2)}_1$ → $z'^{(2)}_1$ → $a'^{(2)}_1$ → $a^{(2)}_1$ second hidden layer

...

# Optimization

# Batch and Stochastic Gradient Descent



SGD and its variants are used almost universally in deep network training.

# Stochastic Gradient Descent updates

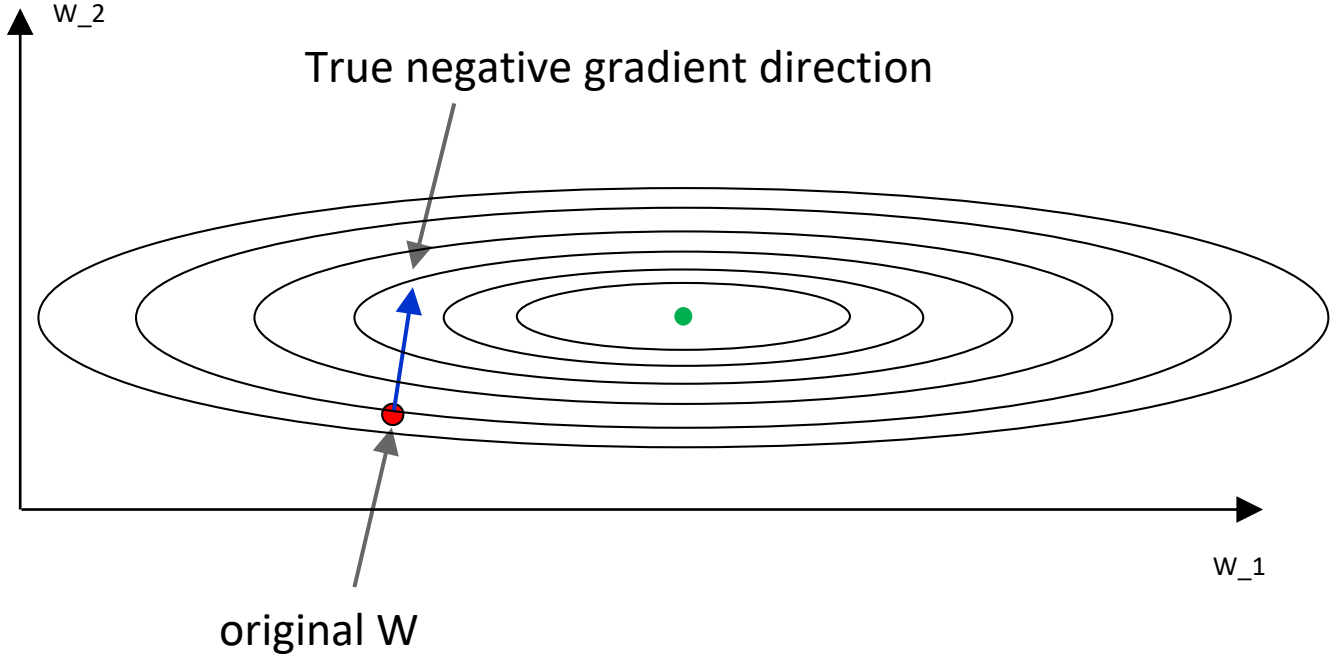
SGD uses $g^{(t)}$, the gradient of a minibatch instead of the true average gradient (call it $g$) on the full dataset. $g = \mathbb{E}\left[g^{(t)}\right]$
(expectation is over minibatches sampled from the full dataset, so that $g^{(t)}$ is an unbiased estimate of $g$. )

But $g^{(t)}$ will typically have a lot of variance (is noisy) compared to $g$

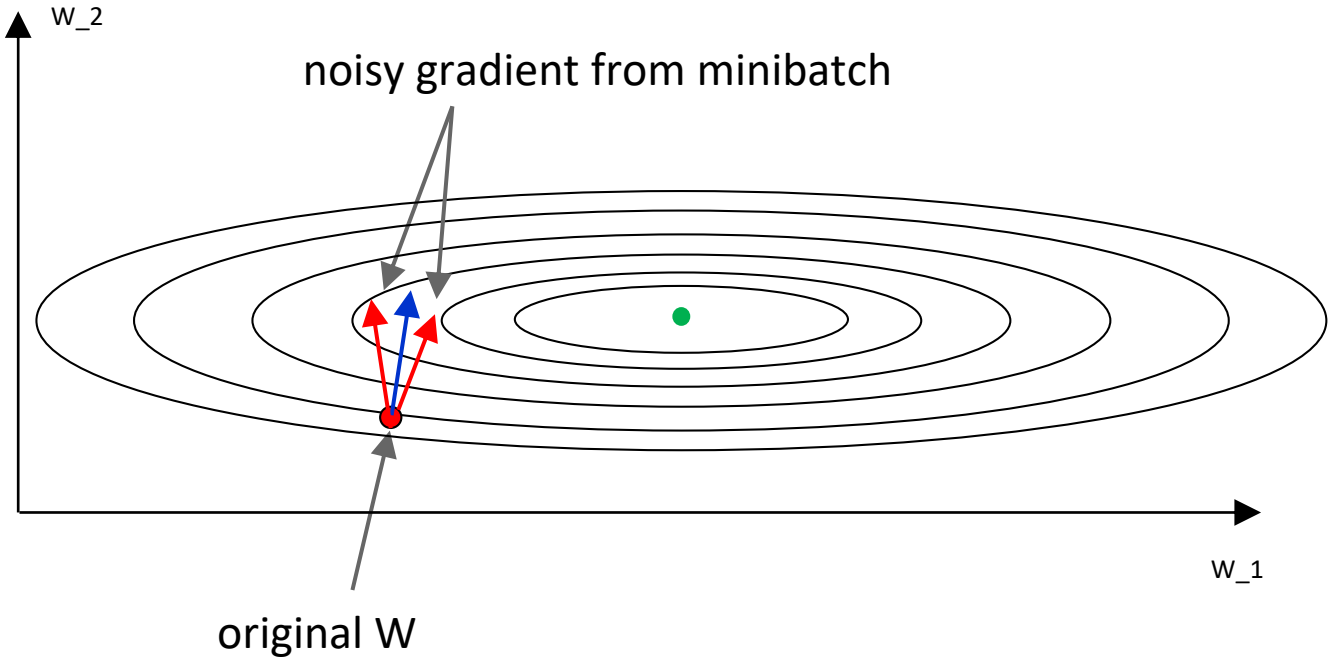
SGD is “stochastic” because it uses $g^{(t)}$ instead of the true gradient $g$

# Minibatch updates

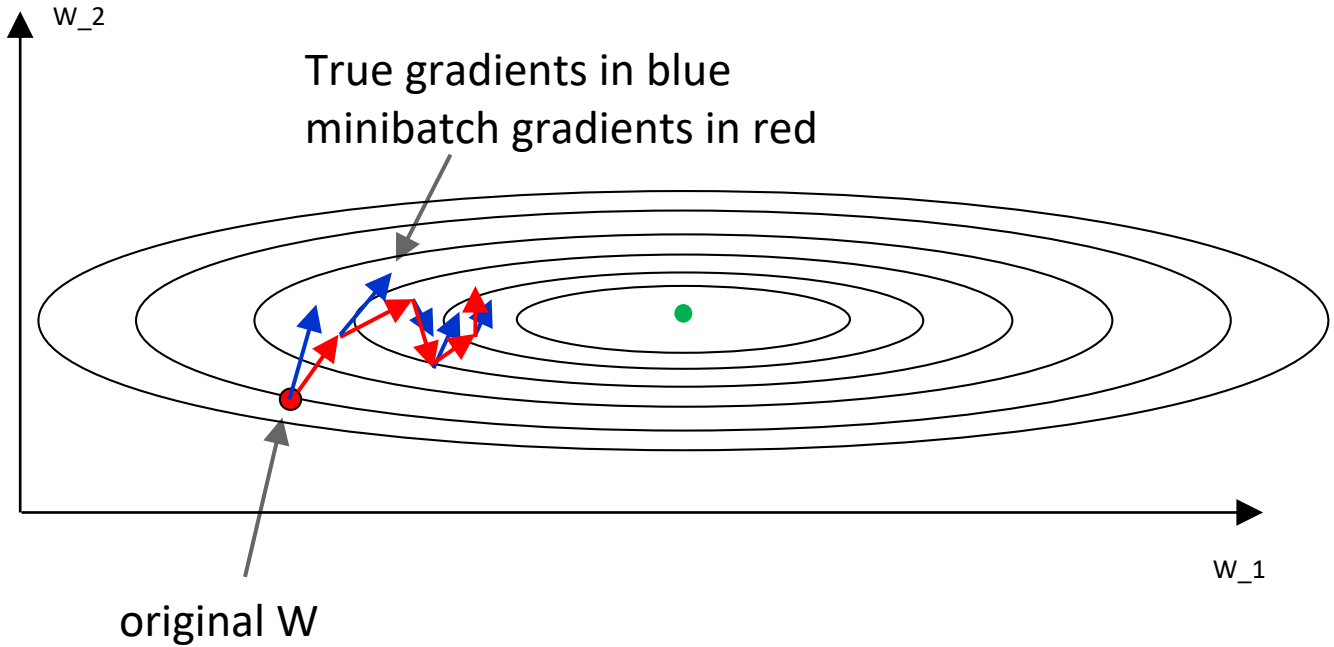
A contour plot represents a function with contours of equal value $f(x) = c$ .
The gradient of the function is always orthogonal to the contour.



# Stochastic Gradient

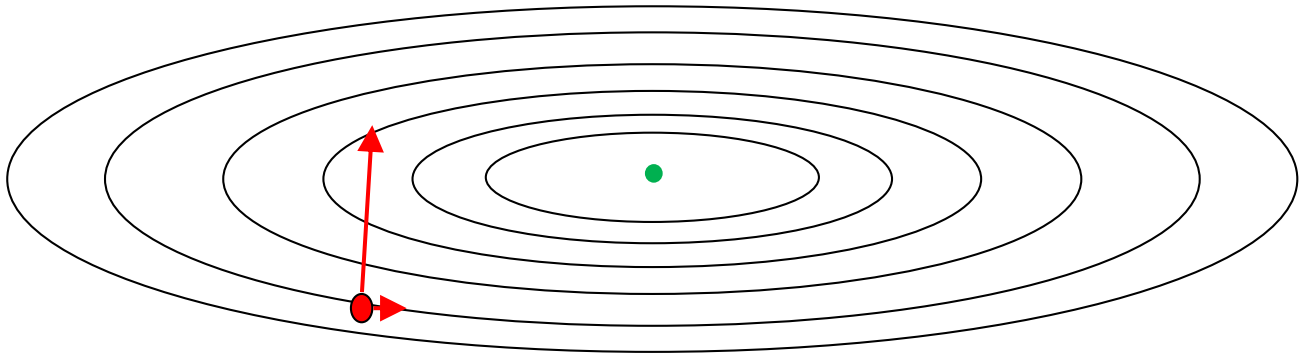


# Stochastic Gradient Descent



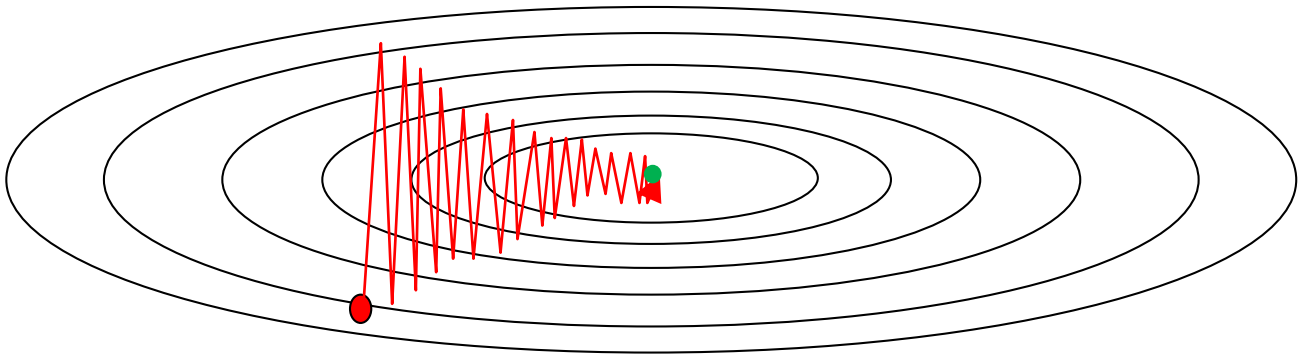

Gradients are noisy but still make good progress on average

# Gradient ≠ Best direction to step



What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction

# Gradient ≠ Best direction to step

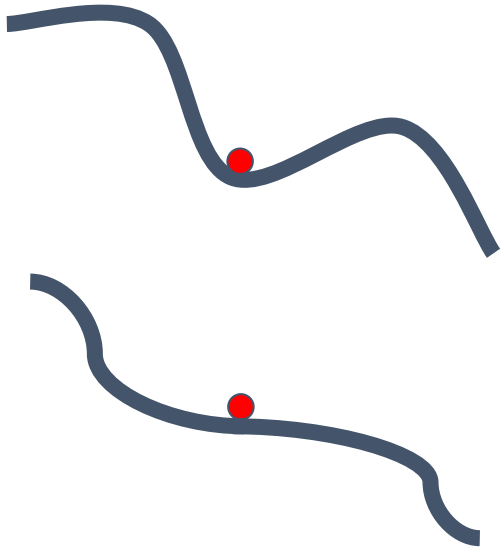


The gradient is much larger across the bowl, so the directions of the gradient steps tends to be vertical.

# Optimization: Problems with SGD

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck





Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

# Newton's First Law

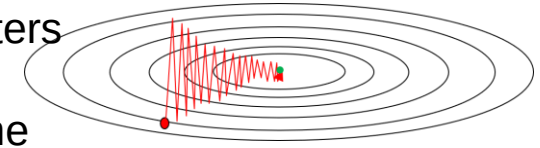
"Every body persists in its state of being at rest or of moving uniformly straight forward, except insofar as it is compelled to change its state by force impressed"

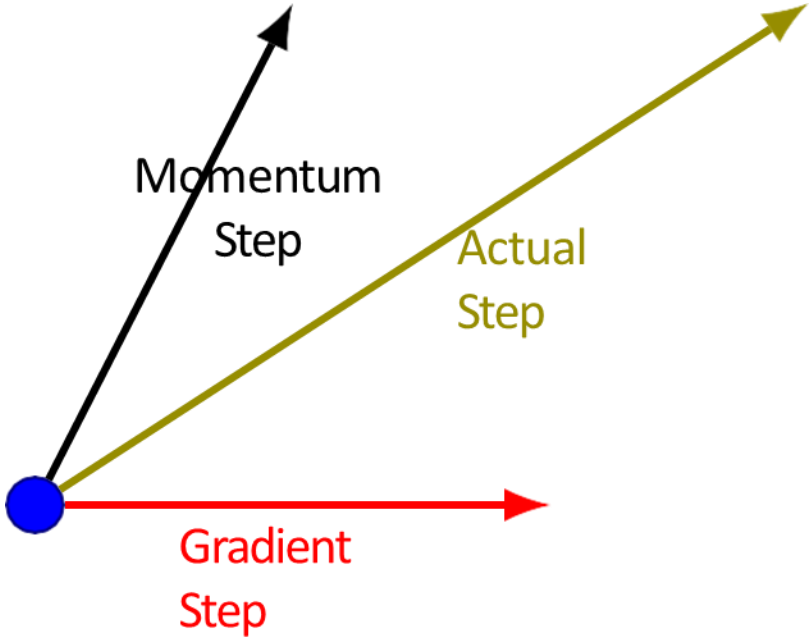
– Isaac Newton

The object's "memory" of its motion state is ***momentum***.

# SGD with Momentum and Damping

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



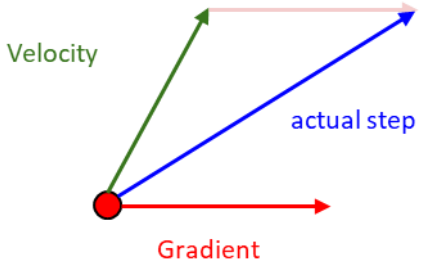


# SGD with Momentum

## SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x -= learning_rate * dx
```




## SGD+Momentum

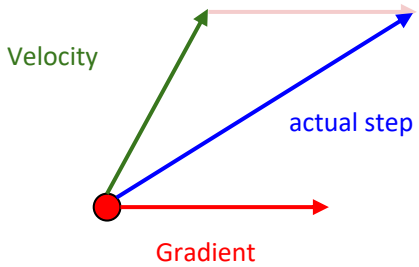
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

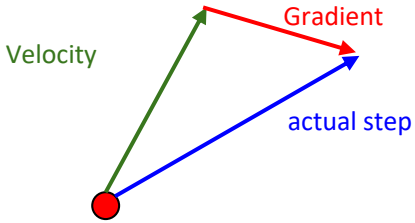
# Nesterov Momentum

Momentum update:




Combine gradient at current point with velocity to get step used to update weights

Nesterov Momentum




"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983
Nesterov, "Introductory lectures on convex optimization: a basic course", 2004
Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

# Nesterov Momentum

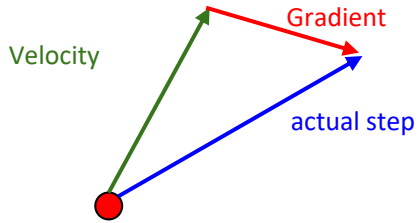
Ordinary Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$




"Look ahead" to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction



Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

# Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$v_{t+1} = \rho v_t - \alpha \nabla f(\tilde{x}_t)$$
$$\tilde{x}_{t+1} = \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1}$$
$$= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t)$$

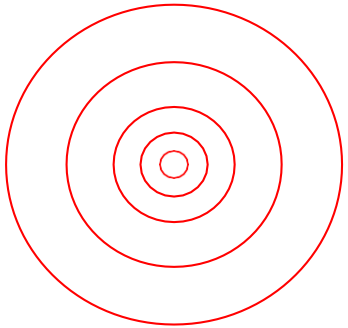
```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```



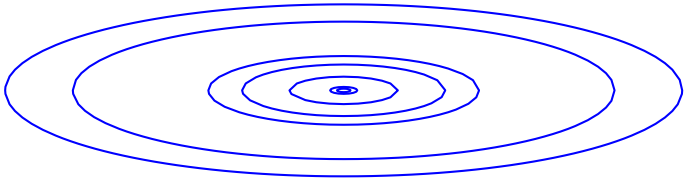
Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

# Adaptive Learning Rate Methods

- Till now we assigned 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.
- Gradient update depends on history of magnitude of gradients
- Parameters with small/sparse updates have larger learning rates
- Downscale a model parameter by square-root of sum of squares of all its historical values

$$\theta_t = \theta_{t-1} - \alpha \nabla_{\theta_{t-1}} f(\theta_{t-1})$$

$$\begin{aligned} g_t &= \sum_{\tau=1}^{t-1} (\nabla f(\theta_\tau))^2 \\ &= g_{t-1} + (\nabla f(\theta_{t-1}))^2 \end{aligned}$$

$$\theta_t = \theta_{t-1} - \frac{\alpha}{\sqrt{g_t + \epsilon}} \odot \nabla f(\theta_{t-1})$$

# RMSProp

[Hinton et al., 2012]

Gradients can vary wildly even though parameters often have the same scale.

RMSprop scales the gradients by the inverse of a moving average, RMS (Root-Mean-Squared) gradient. Define

$$s^{(t)} = \beta s^{(t-1)} + (1-\beta)\left(g^{(t)}\right)^2$$

where $s^{(t)}$ is the (moving average) Mean-Squared Gradient at step $t$,

$g^{(t)}$ is the normal minibatch gradient at step $t$,

$\beta \in [0,1]$ is a moving-average decay factor, (close to 1)

$\left(g^{(t)}\right)^2$ is the element-wise square of $g^{(t)}$, so $s^{(t)}$ has same dims as $g^{(t)}$.

**RMSprop:**

$$W^{(t+1)} = W^{(t)} - \alpha \frac{g^{(t)}}{\sqrt{s^{(t)}}}$$

# Adam

- RMSProp and Momentum take contrasting approaches.
  - momentum accelerates our search in direction of minima
  - RMSProp impedes our search in direction of oscillations.

- Adam or Adaptive Moment Optimization algorithms combines the heuristics of both Momentum and RMSProp

# Momentum + RMSprop ≈ ADAM

[Kingma and Ba, 2014]

- Compute moving averages of the gradient and squared gradient.
- Treat them as moments and add a small-sample bias correction (next slide):

$$p^{(t)} = \beta_1 p^{(t-1)} + (1-\beta_1) g^{(t)}$$
$$s^{(t)} = \beta_2 s^{(t-1)} + (1-\beta_2)\left(g^{(t)}\right)^2$$

- Then normalize the momentum update (no bias correction):

$$W^{(t+1)} = W^{(t)} - \alpha \frac{p^{(t)}}{\sqrt{s^{(t)}}}$$

- Important practical point: $\beta_1$ typically 0.9, $\beta_2$ typically much closer to 1, e.g. 0.9999

# Adam

One of the most popular learning algorithms.

- Adaptive learning rate as RMSprop, but with momentum & correction bias
- Maintains an exponentially decaying average of past squared gradients $v_t$
- An exponentially decaying average of past gradients $m_t$ (similar to momentum)
- 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

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

0.9 for β1
0.999 for β2
$\epsilon = 10^{-8}$.