بسم الله الرحمن الرحیم

Deep Learning

Mohammad Reza Mohammadi

2021

Stochastic Gradient Descent (SGD)

$$L = \frac{1}{N}\sum_{i=1}^{N} L_i(s_i, y_i)$$

$$\nabla_W L = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(s_i, y_i)$$

$$s_i = f(x_i, W\ ) = W\ x_i$$

$$W = W - \eta \nabla_W L$$

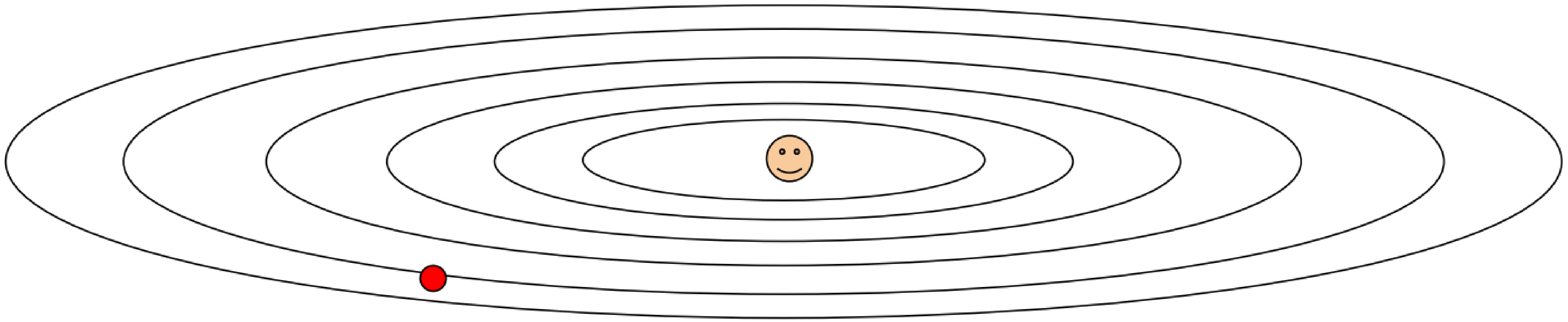
- Full sum expensive when N is large!
- Approximate sum using a minibatch of examples
  - 32 / 64 / 128 common

```python
# Vanilla Minibatch Gradient Descent

while True:
  data_batch = sample_training_data(data, 256) # sample 256 examples
  weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
  weights += - step_size * weights_grad # perform parameter update
```

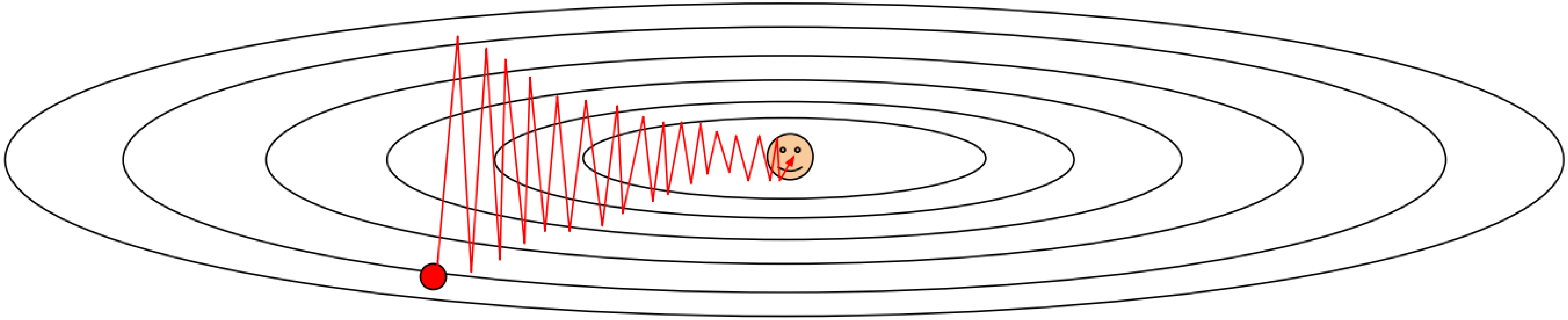
Problems with SGD

- What if loss changes quickly in one direction and slowly in another?
- What does gradient descent do?



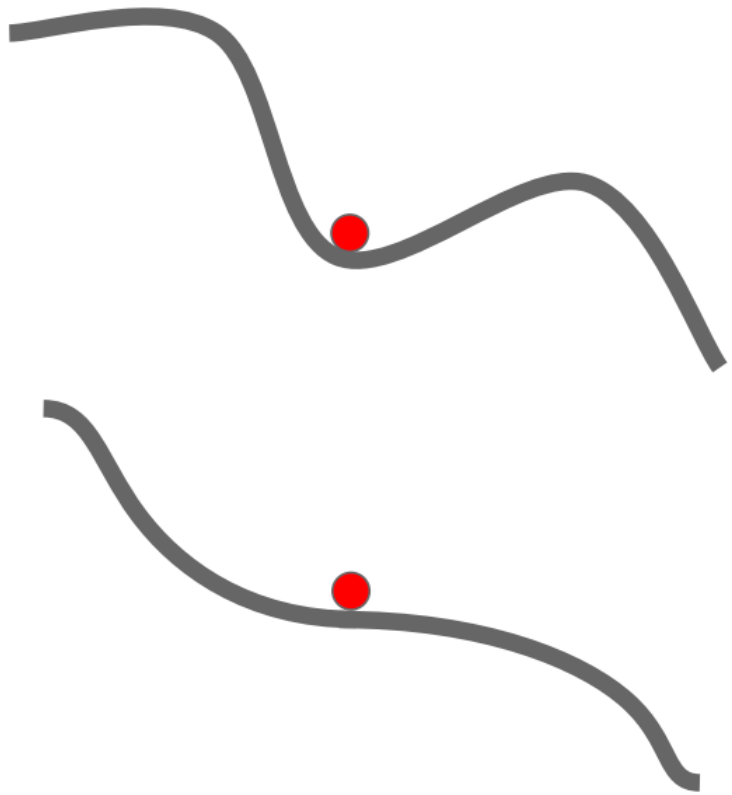
Problems with SGD

- 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



Problems with SGD

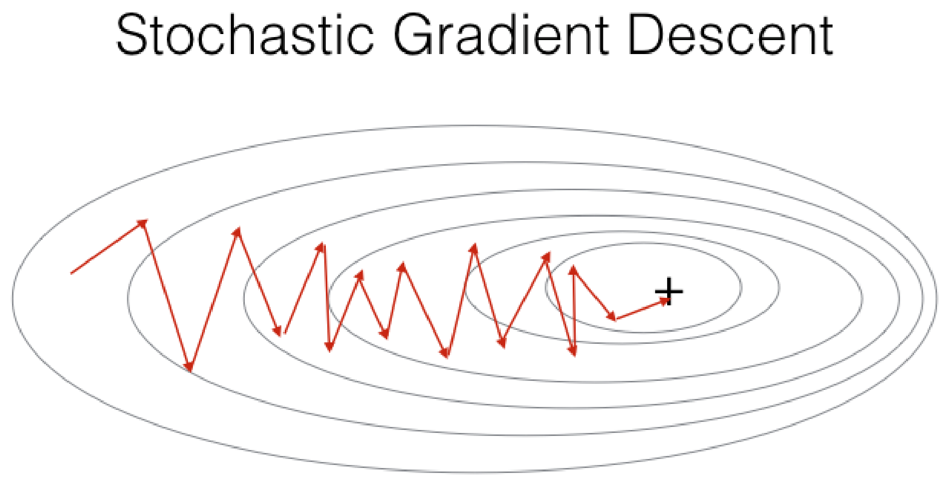
- What if the loss function has a local minima or saddle point?
- Zero gradient, gradient descent gets stuck

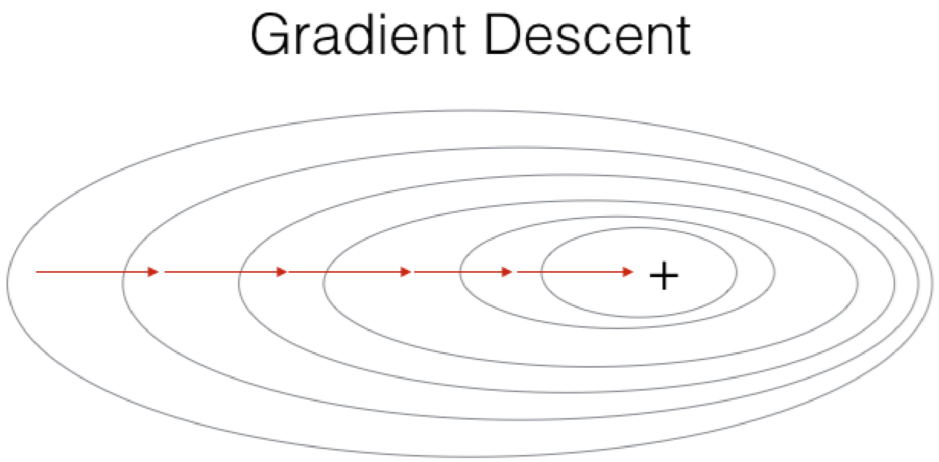


Problems with SGD

- Our gradients come from minibatches so they can be noisy!
- Gradients are noisy but still make good progress on average

$$\nabla_W L = \frac{1}{N}\sum_{i=1}^{N} \nabla_W L_i(s_i, y_i)$$





SGD + Momentum

- The loss can be interpreted as the height of a hilly terrain
- Initializing the parameters with random numbers is equivalent to setting a particle with zero initial velocity at some location
- The optimization process can then be seen as equivalent to the process of simulating the parameter vector (i.e. a particle) as rolling on the landscape

SGD + Momentum

SGD

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

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

SGD + Momentum

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

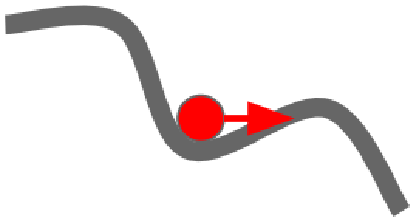
```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x -= learning_rate * vx
```

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

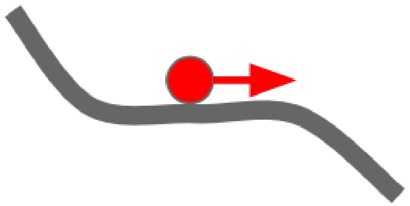
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically $\rho = 0.9$ or 0.99

SGD + Momentum

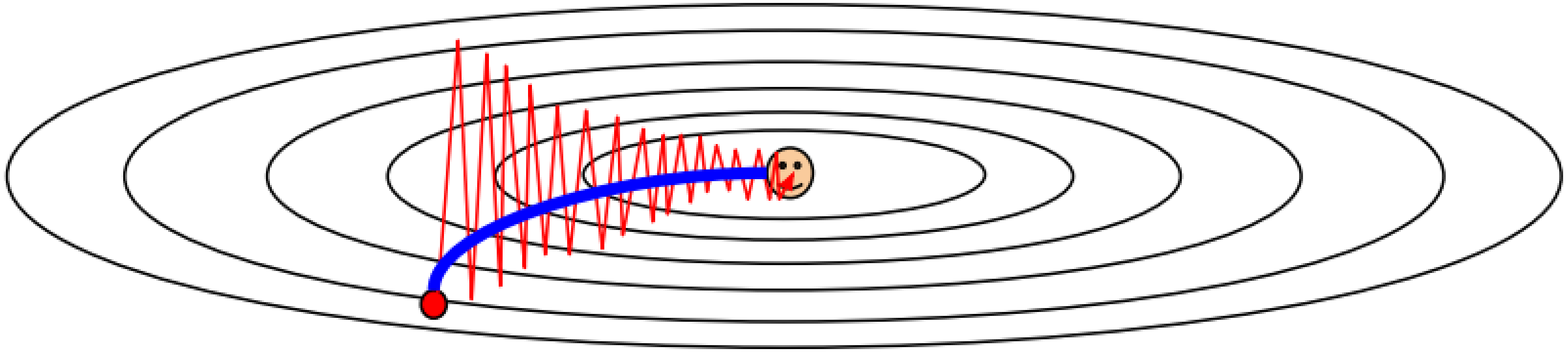
Local Minima

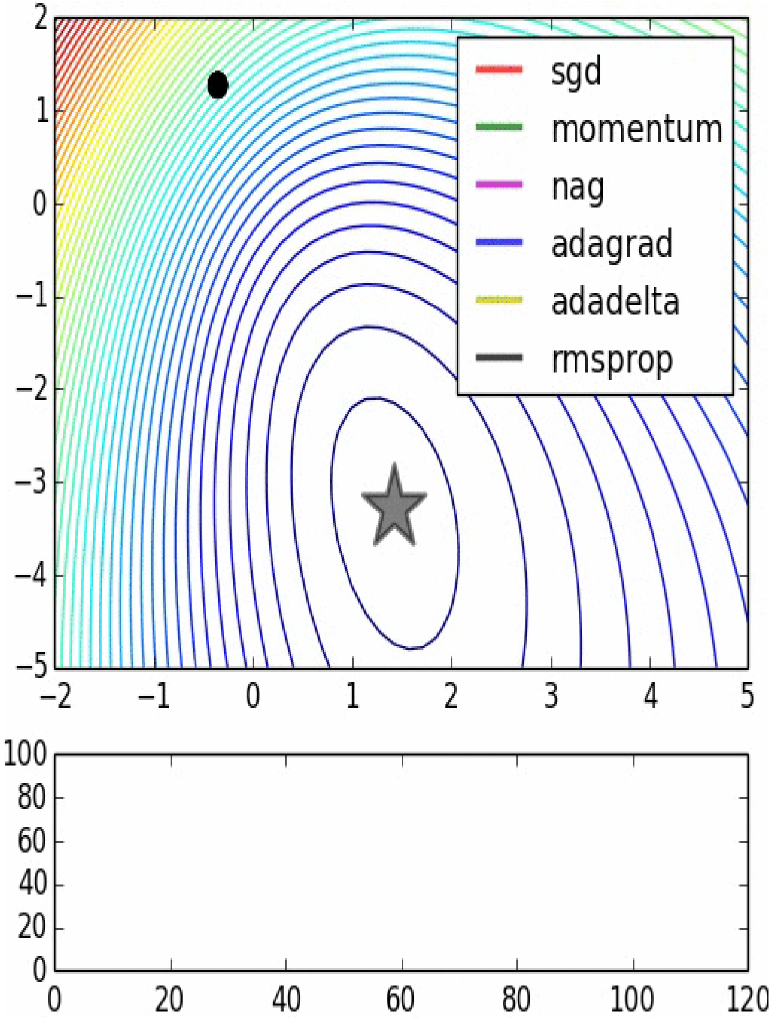


Saddle points



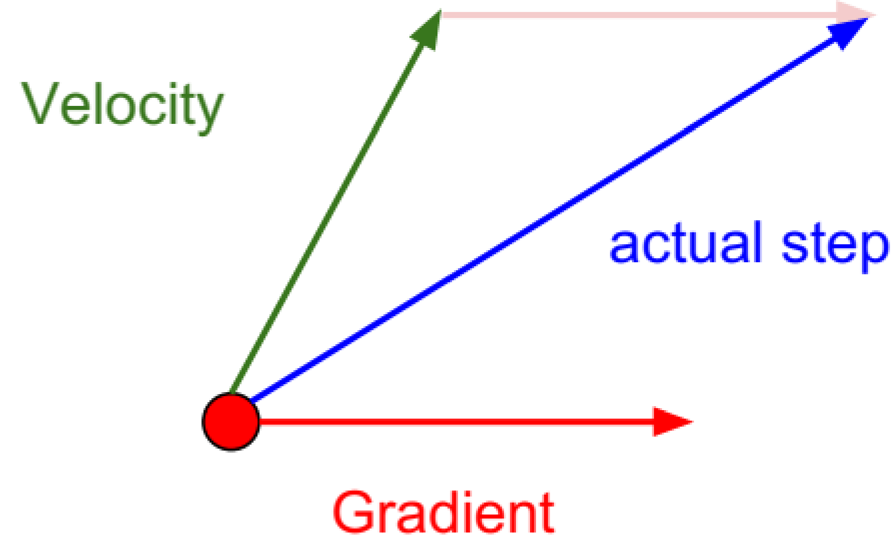
Poor Conditioning



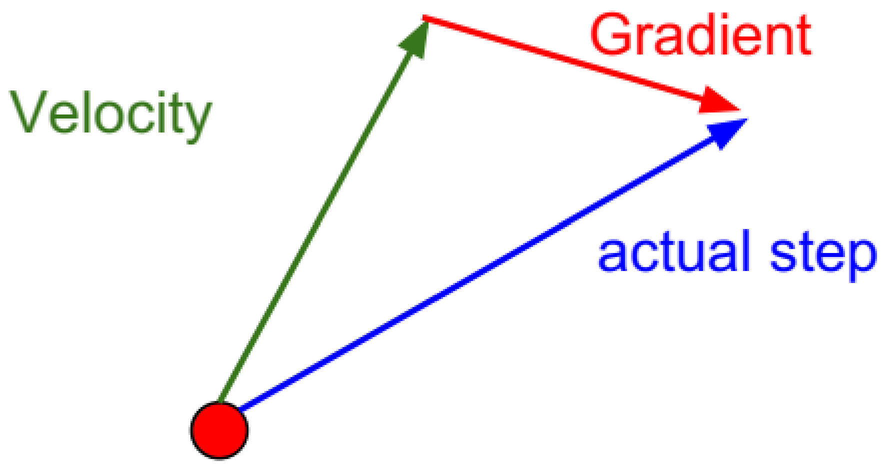


Nesterov Momentum

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



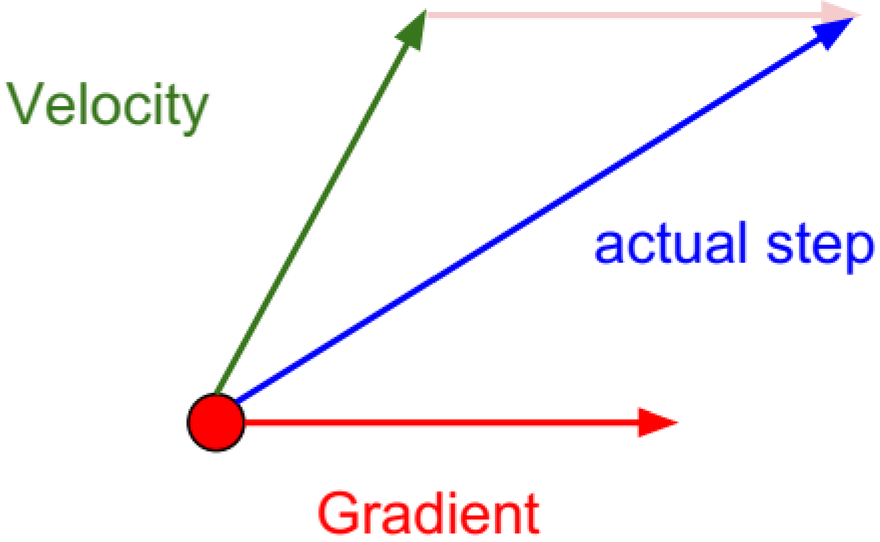
- Nesterov Momentum: “Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

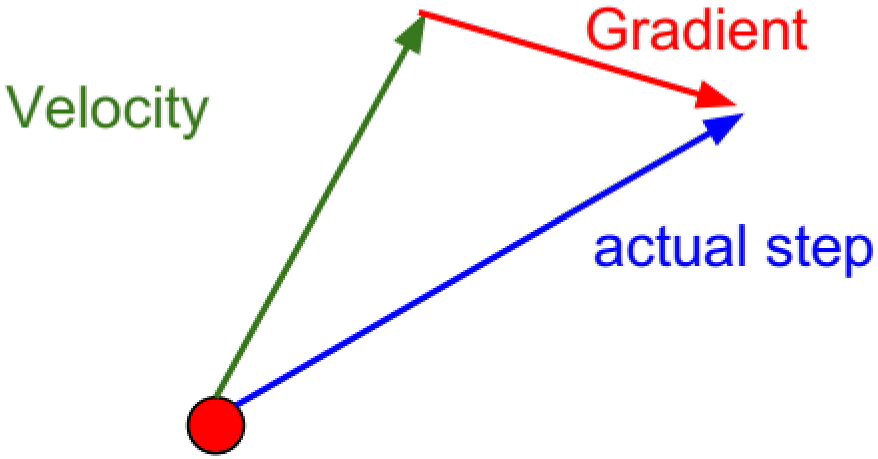


Nesterov Momentum

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

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





Nesterov Momentum

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

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

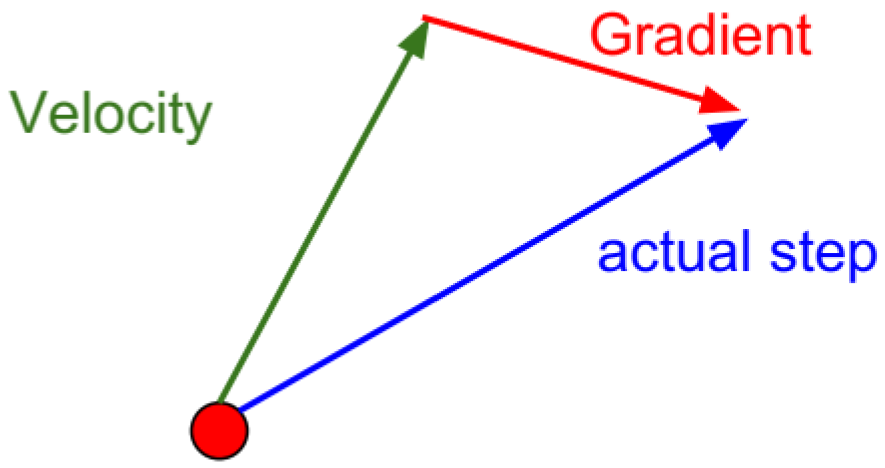
- Change of variables $\tilde{x}_t \triangleq x_t + \rho v_t$

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

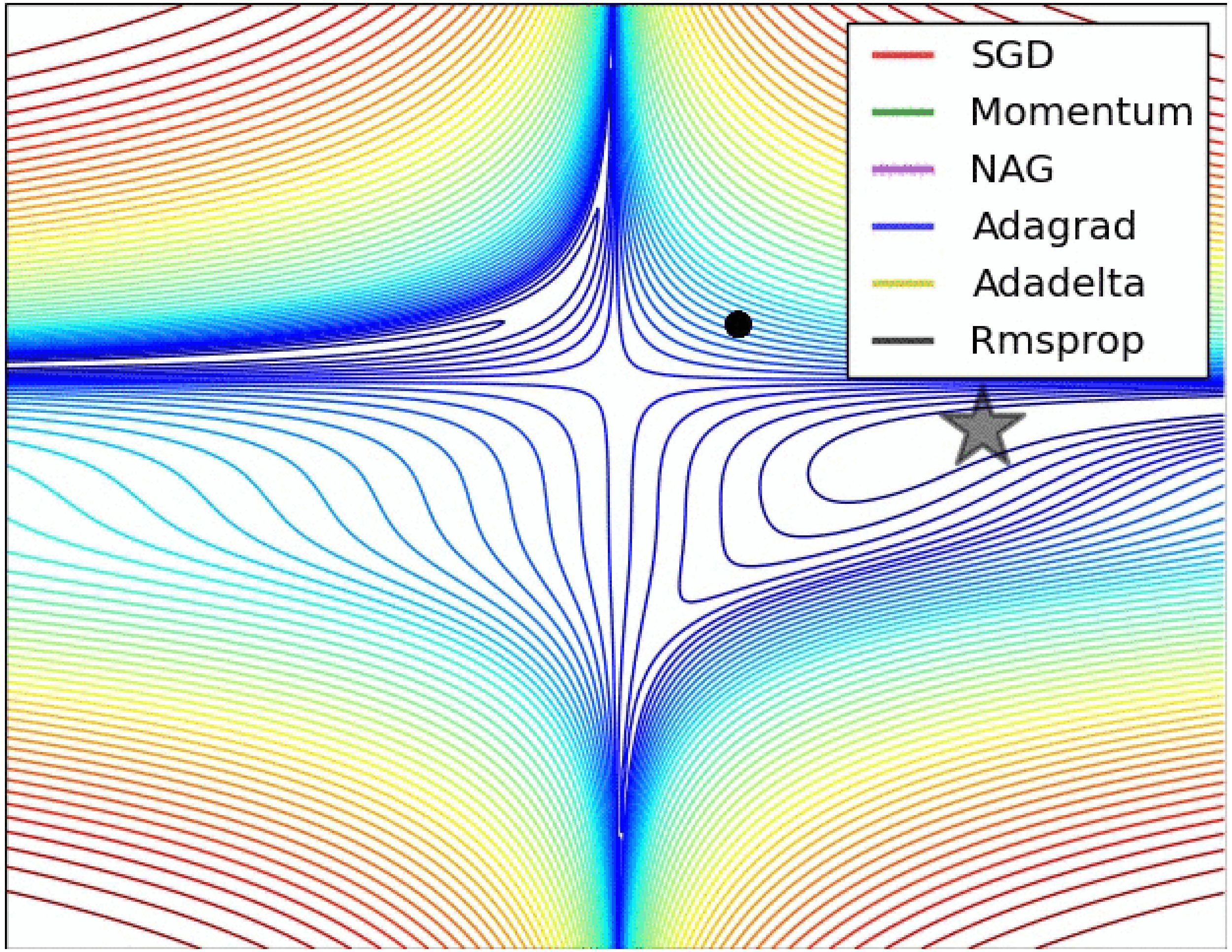
$$\tilde{x}_{t+1} - \rho v_{t+1} = \tilde{x}_t - \rho v_t + v_{t+1}$$

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

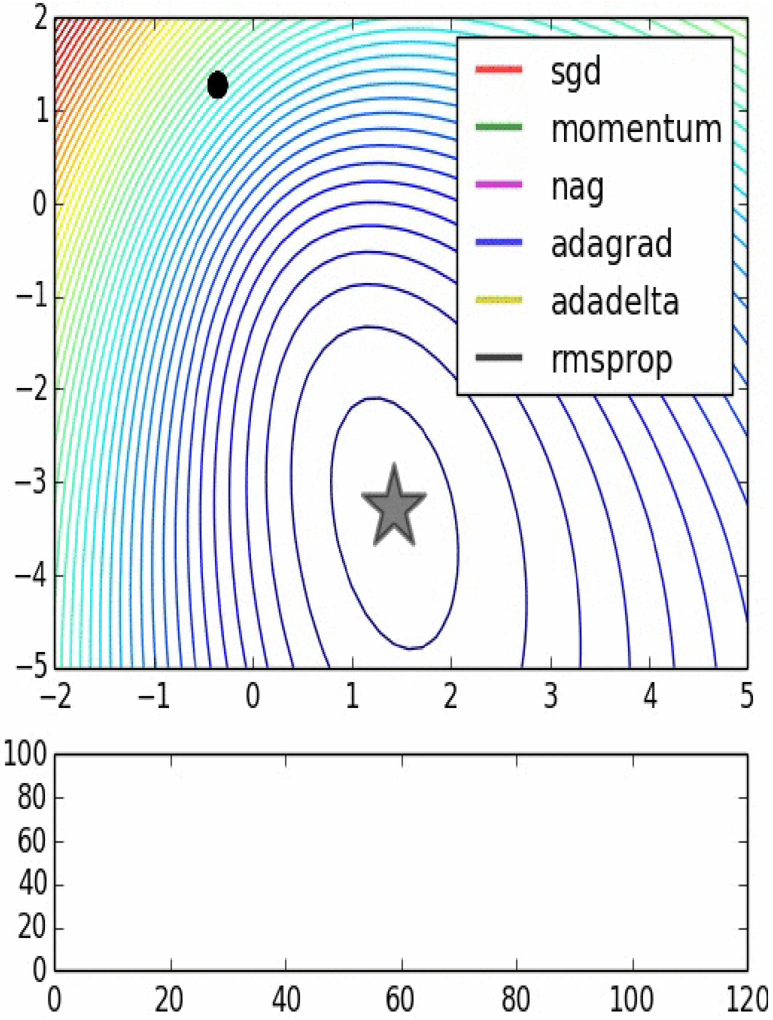
```
dx = compute_gradient(x)
old_v = v
v = rho * v - learning_rate * dx
x += -rho * old_v + (1 + rho) * v
```



Nesterov Momentum



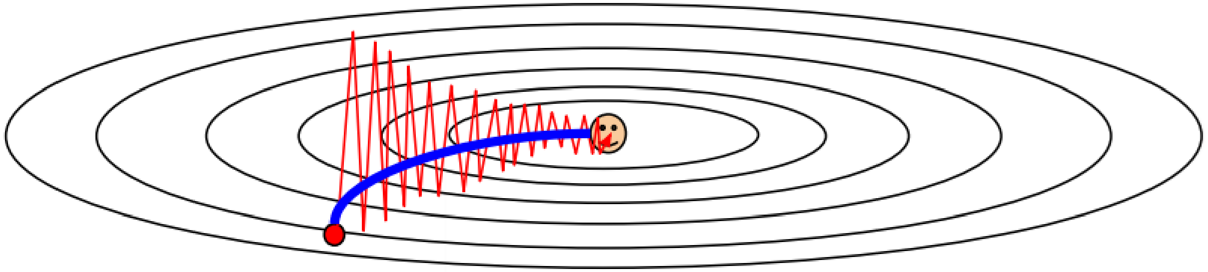
(image credits to Alec Radford)



AdaGrad

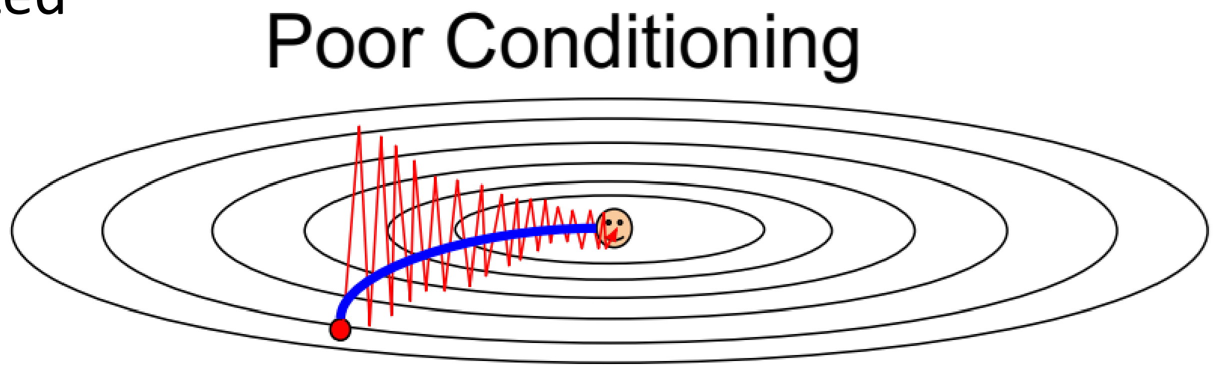
- Added element-wise scaling of the gradient based on the historical sum of squares in each dimension
- "Per-parameter learning rates" or "adaptive learning rates"

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



AdaGrad

- Added element-wise scaling of the gradient based on the historical sum of squares in each dimension
- “Per-parameter learning rates” or “adaptive learning rates”
  - Progress along “steep” directions is damped
  - progress along “flat” directions is accelerated
  - step size over long time decays to zero




```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

AdaGrad / RMSProp

- First and second moments will be very small at the initial timesteps

Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

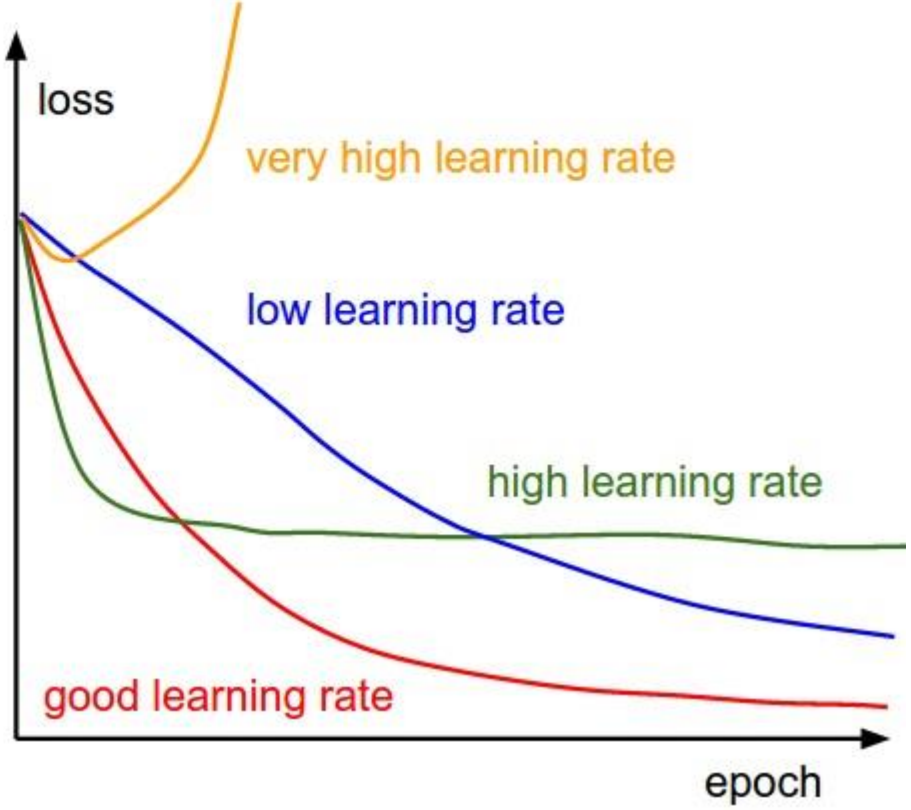
Bias correction

AdaGrad / RMSProp

- Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

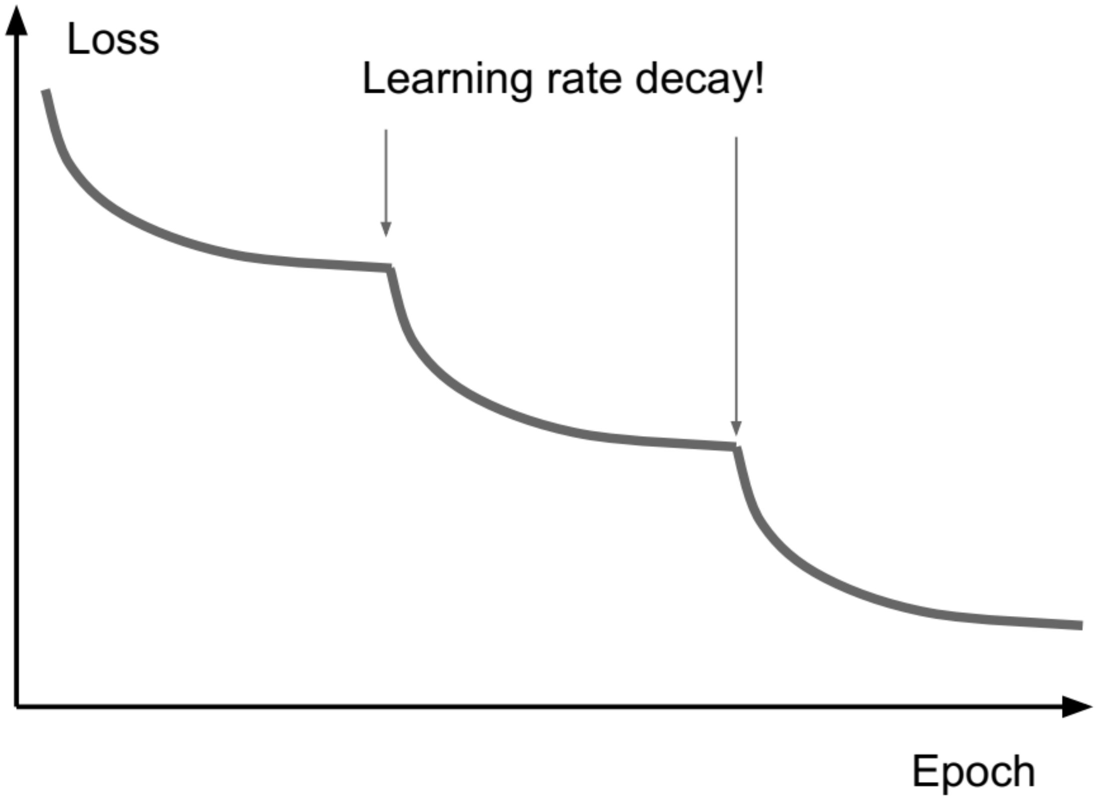
Learning rate

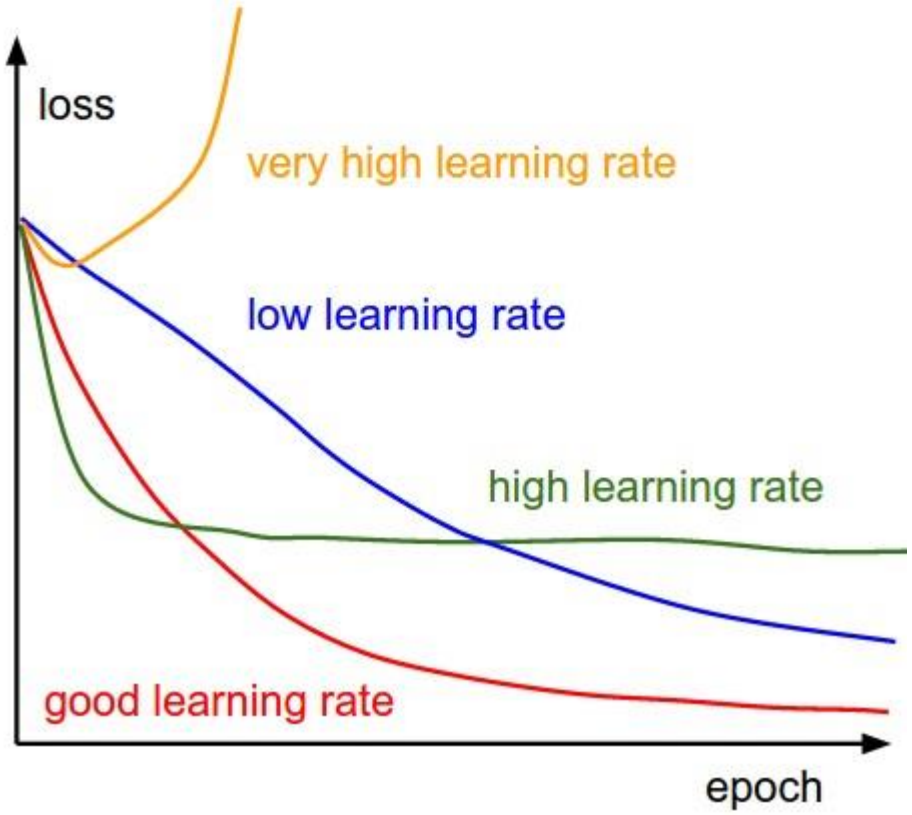
- SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have learning rate as a hyperparameter
- We can decay learning rate over time:
  - step decay:
    - e.g. decay learning rate by half every few epochs
  - exponential decay:
    - $\alpha = \alpha_0 e^{-kt}$
  - 1/t decay:
    - $\alpha = \alpha_0 / (1 + kt)$



Learning rate decay

- More critical with SGD+Momentum, less common with Adam





Keras Callbacks

Using custom callbacks

Creating new callbacks is a simple and powerful way to customize a training loop. Learn more about creating new callbacks in the guide Writing your own Callbacks, and refer to the documentation for the base `Callback` class.

Available callbacks

- Base Callback class
- ModelCheckpoint
- TensorBoard
- EarlyStopping
- LearningRateScheduler
- ReduceLROnPlateau
- RemoteMonitor
- LambdaCallback
- TerminateOnNaN
- CSVLogger
- ProgbarLogger

https://colab.research.google.com/drive/1HRYZrbRLuNPHQdKWEkUUR9atkN5sy0SF?usp=sharing