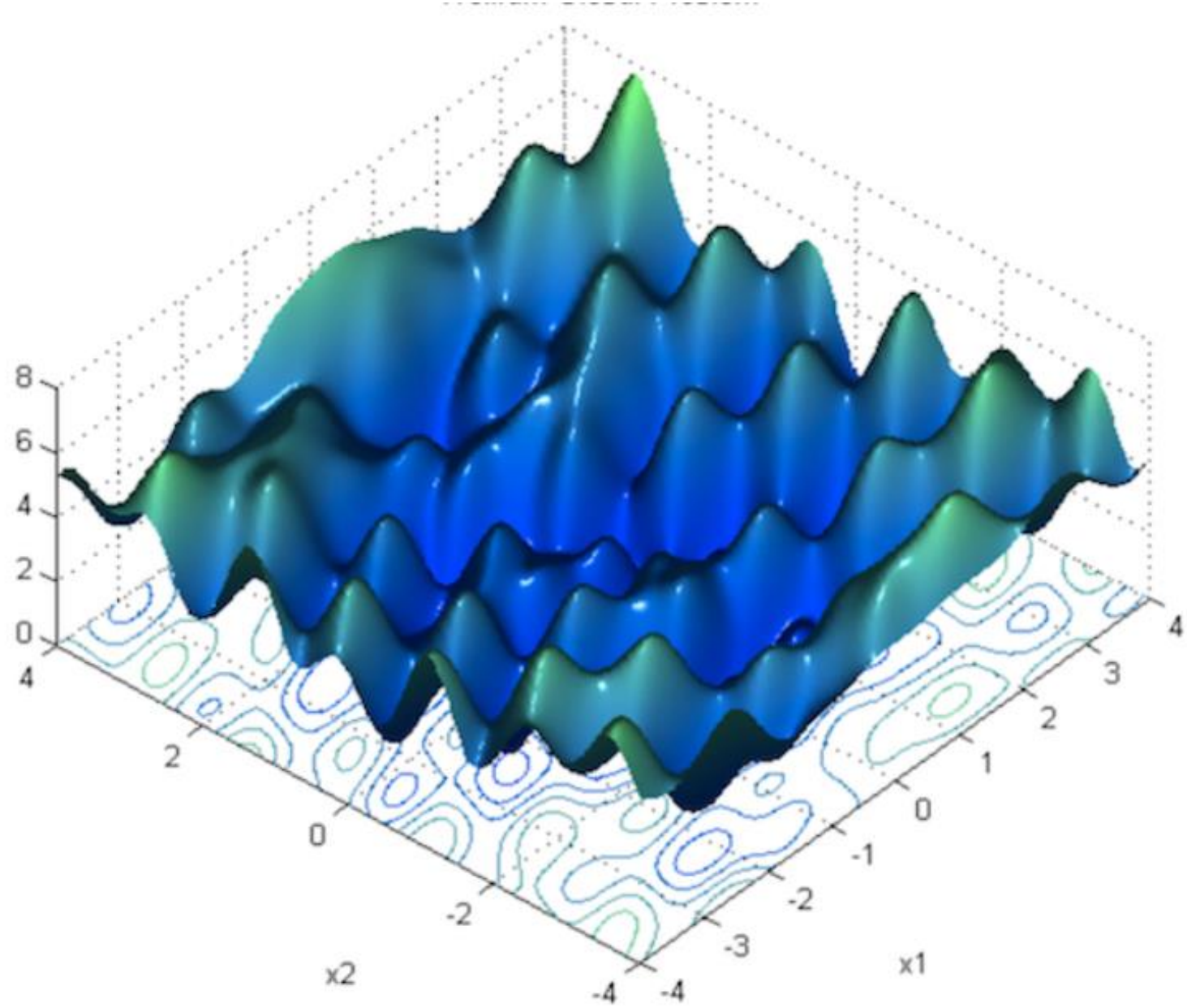
OPTIMIZATION FOR DNN

- **Difficulty in finding best parameters' values**
- **SGD**
- **SGD + Momentum**
- **Adaptive learning rates (Adaptive methods)**
- **Adam (Momentum + Adaptive rates)**
- **Second order methods**
- **More on gradient vanishing and exploding**
- **Training of LSTM**

- Combine materials from Fei-Fei Lee ppt, Chap. 8 of Deep Learning (Goodfellow, etc.) and how neural networks are trained

Difficulty in finding optimal parameter set in DNN

- Too many parameters, hundreds of thousand to million
- Curse of dimensionality– random search or grid search are impossible
- The convexity of the loss function is very complex, many local optimums, saddle points—for d parameters, when gradients are all zero, there is only $\frac{1}{3^d}$ possibility to be an optimum, and it could be a local minimum, or a local maximum, not the global minimum
- You may find a local minimum



Example of non-convex loss surface with two parameters. Note that in deep neural networks, we're dealing with millions of parameters, but the basic principle stays the sam. Source: Yoshua Bengio.

SGD (Stochastic Gradient Descent)

- Mini-batch gradient descent (MB-GD), in which the whole dataset is randomly subdivided into N equally-sized mini-batches of K samples each. K may be a small positive number, or it can be in the dozens or hundreds; it depends on the specific architecture and application. Note that if K=1, then you have SGD, and if K is the size of the whole dataset, it is **batch** gradient descent. Note also that confusingly, sometimes people say "SGD" to refer to both MB-GD and **one sample at a time**.

Mini-batch SGD

Algorithm 8.1 Stochastic gradient descent (SGD) update At training iteration k

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

Require: Initial parameter $\boldsymbol{\theta}$

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$

$k \leftarrow k + 1$

end while

Problem with SGD

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

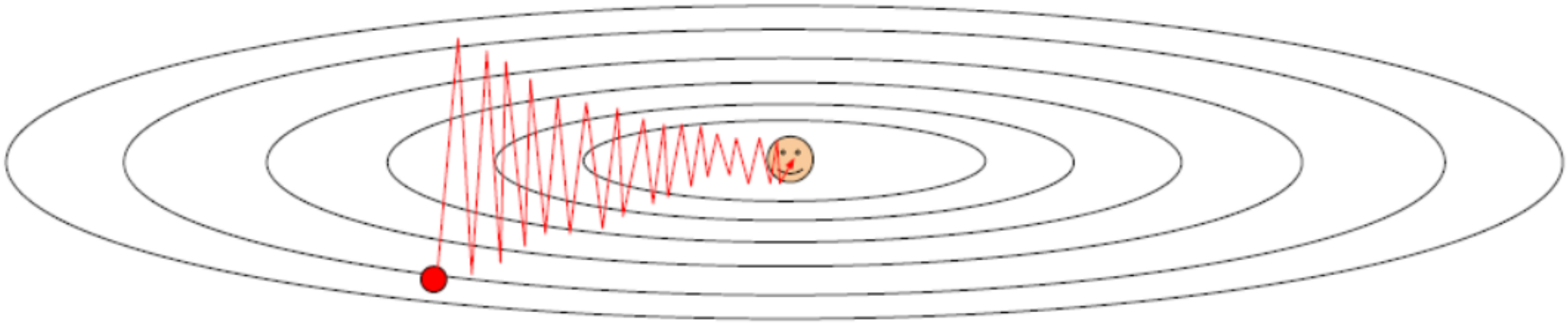
```
# Vanilla Gradient Descent

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

Lee’s

Optimization: Problems with SGD

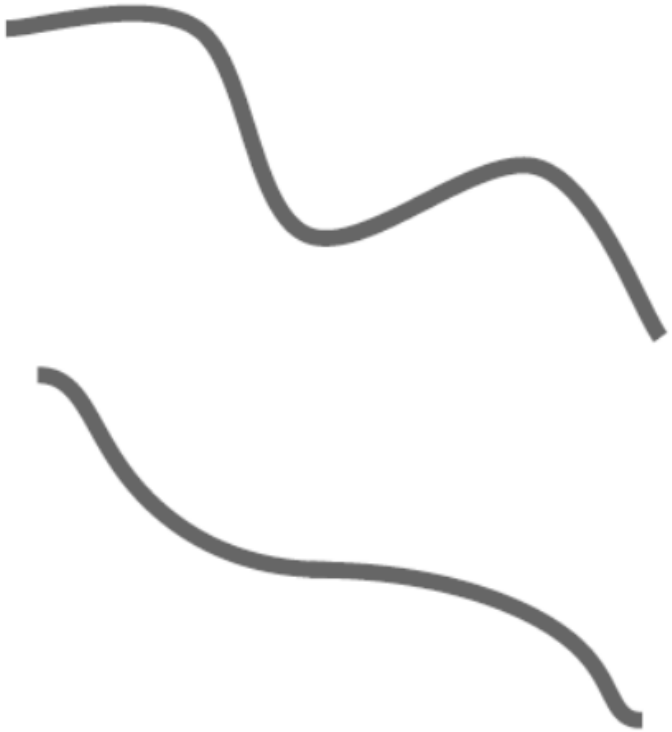
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Lee's

Optimization: Problems with SGD

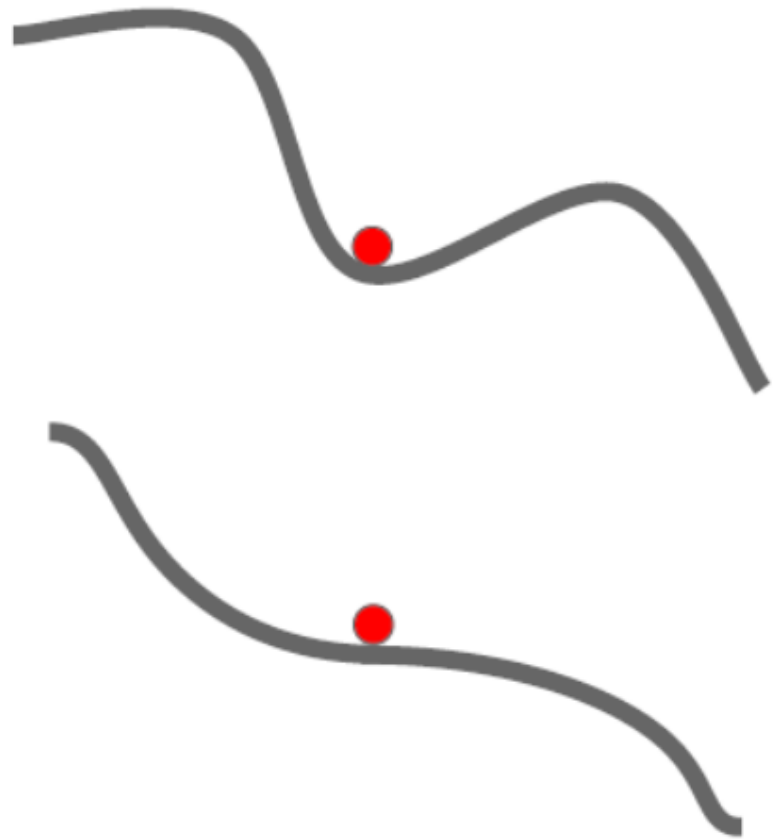
What if the loss function has a **local minima** or **saddle point**?



Optimization: Problems with SGD

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

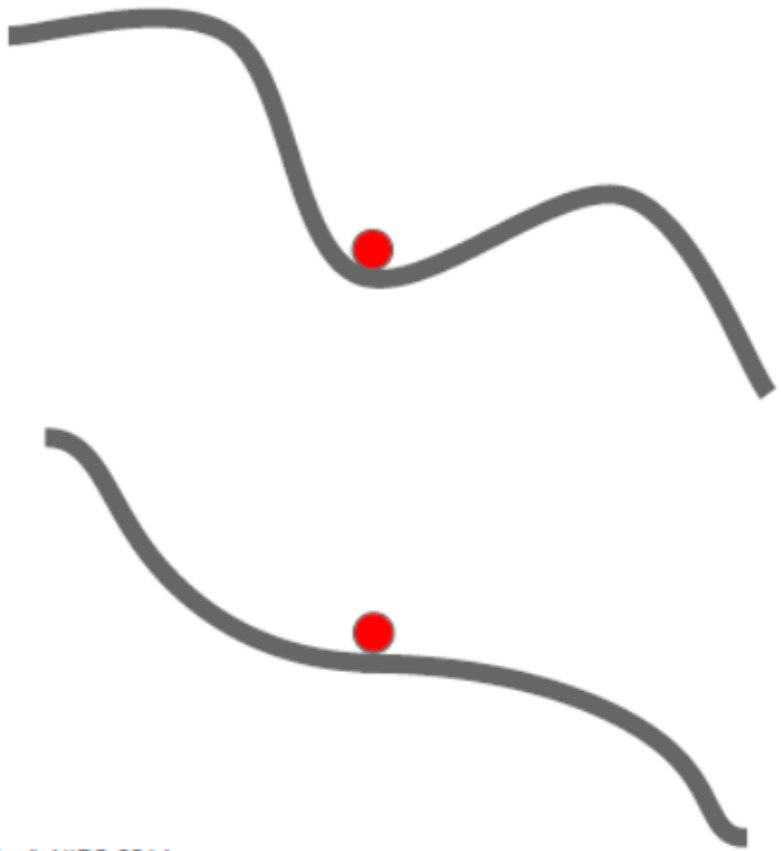
Zero gradient, gradient descent gets stuck



Optimization: Problems with SGD

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

Saddle points much more common in high dimension



Solution 1--add momentum

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
```

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

SGD + Momentum

SGD+Momentum

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

$$x_{t+1} = x_t + v_{t+1}$$

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

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
```

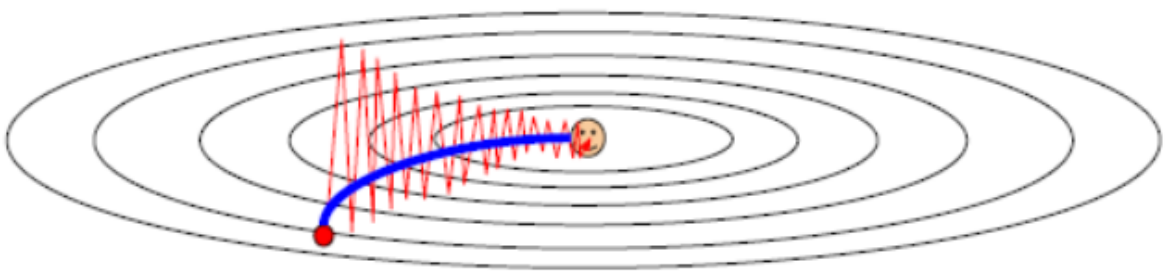
You may see SGD+Momentum formulated different ways, but they are equivalent - give same sequence of x

SGD + Momentum

Local Minima

Saddle points

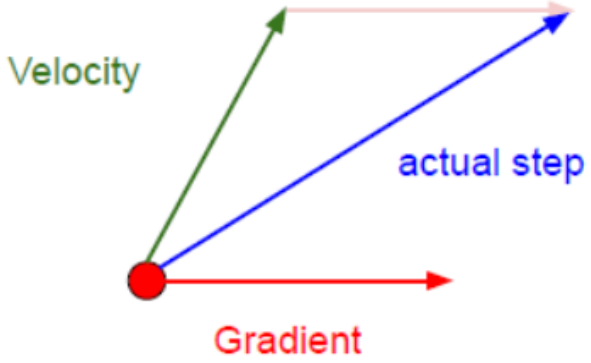
Poor Conditioning



Nesterov's momentum accelerated gradient(NAG)

Nesterov Momentum

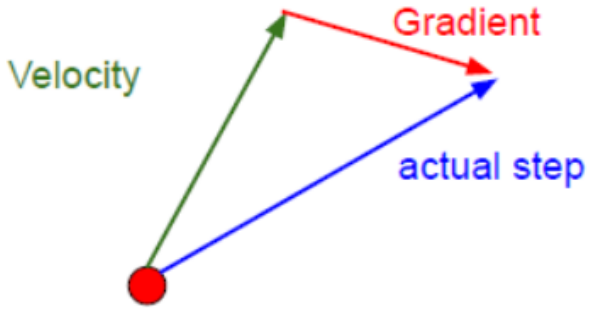
Momentum update:



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

Nesterov, "A method of solving a convex programming problem with convergence rate O(1/k^2)", 1983

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

Use mini-batch

$$\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \nabla_{\boldsymbol{\theta}} \left[\frac{1}{m} \sum_{i=1}^{m} L\left(\boldsymbol{f}(\boldsymbol{x}^{(i)}; \boldsymbol{\theta} + \alpha \boldsymbol{v}), \boldsymbol{y}^{(i)} \right) \right], \tag{8.21}$$

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}, \tag{8.22}$$

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.

Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$.

Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

Adaptive learning rate--AdaGrad

- AdaGrad stands for **Adaptive subGradient**. how neural networks are trained
- Adaptive methods tend to scale gradient components differently. It will decrease the component which are always big.

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, perhaps 10^{-7}, for numerical stability

Initialize gradient accumulation variable $\boldsymbol{r} = \boldsymbol{0}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$.

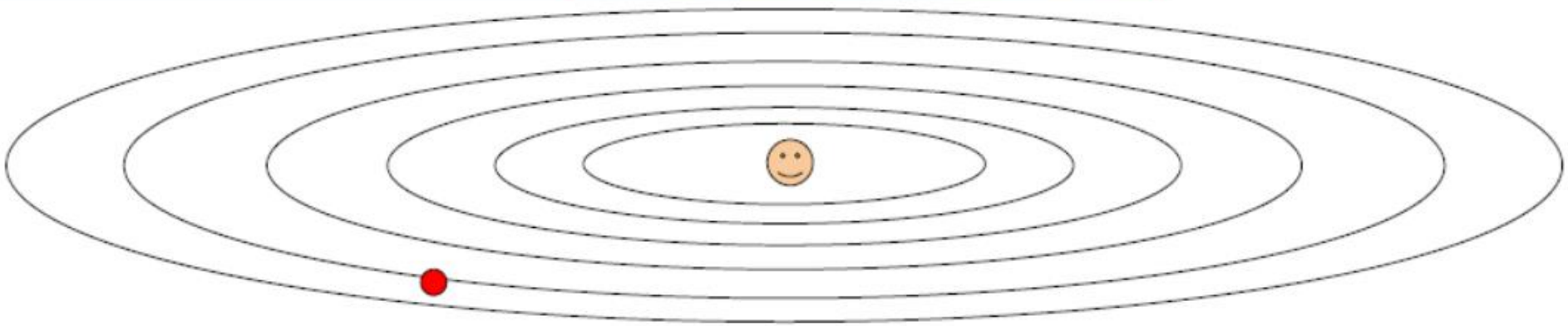
Compute update: $\Delta \boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

AdaGrad

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



Q2: What happens to the step size over long time? Decays to zero

RMSProp: “Leaky AdaGrad”

AdaGrad

```
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)
```

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ

Require: Initial parameter $\boldsymbol{\theta}$

Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers

Initialize accumulation variables $\boldsymbol{r} = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Accumulate squared gradient $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$.

Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$.

end while

ρ controls the length scale of the moving window. RMSProp is effective and practical.

Finally, the Adam-consider momentum and adaptive rate scaling

- Adam stands for “Adaptive Moment”
- First moment for momentum (+current gradient)
- Second moment for weight scaling

Adam

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)

Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)

Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})

Require: Initial parameters $\boldsymbol{\theta}$

Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$

Initialize time step $t = 0$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

$t \leftarrow t + 1$

Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$

Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$

Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$

Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$

Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Reference: Kingma, Diederik and Jimmy Ba, “Adam: A method for Stochastic Optimization

$$v_i = \beta_2 v_{i-1} + (1-\beta_2) g_i \odot g_i$$

$$v_i = \beta_2^{i-1}(1-\beta_2) g_1 \odot g_1 + \beta_2^{i-2}(1-\beta_2) g_2 \odot g_2 + \cdots + (1-\beta_2) g_i \odot g_i$$

$$v_i = (1-\beta_2) \sum_{k=1}^{i} \beta^{i-k} g_k \odot g_k$$

$$E(v_i) = E[(1-\beta_2) \sum_{k=1}^{i} \beta^{i-k} (g_k \odot g_k)]$$

Assuming that, $E[g_k \odot g_k] = E[g_i \odot g_i]$

We have,

Note that, $1 - x^n = (1 - x)(1 + x + x^2 + \cdots + x^{n-1})$

$$E(v_i) = E(1 - \beta_2) \sum_{k=1}^{i} \beta^{i-k} \, g_k \odot g_k) = E[g_i \odot g_i] \, (1 - \beta_2) \sum_{k=1}^{i} \beta^{i-k}$$

$$= E[g_i \odot g_i] \, (1 - \beta_2^i)$$

Therefore,

$$\widetilde{v_i} = \frac{v_i}{(1 - \beta_2^i)}$$

So which one is the best?

- Unfortunately, there is currently no consensus on this point. Schaul et al. suggested that the family of algorithms with adaptive learning rates (RMSProp and AdaDelta, and ADAM) perform fairly robustly.
- No single best algorithm has emerged.

- Currently, the most popular optimization algorithms actively in use include SGD, SGD with momentum, RMSProp, RMSProp with momentum, AdaDelta, and Adam.
- The choice depend one the user's familiarity with the algorithm(for ease of hyperparameter tuning)

Lee's

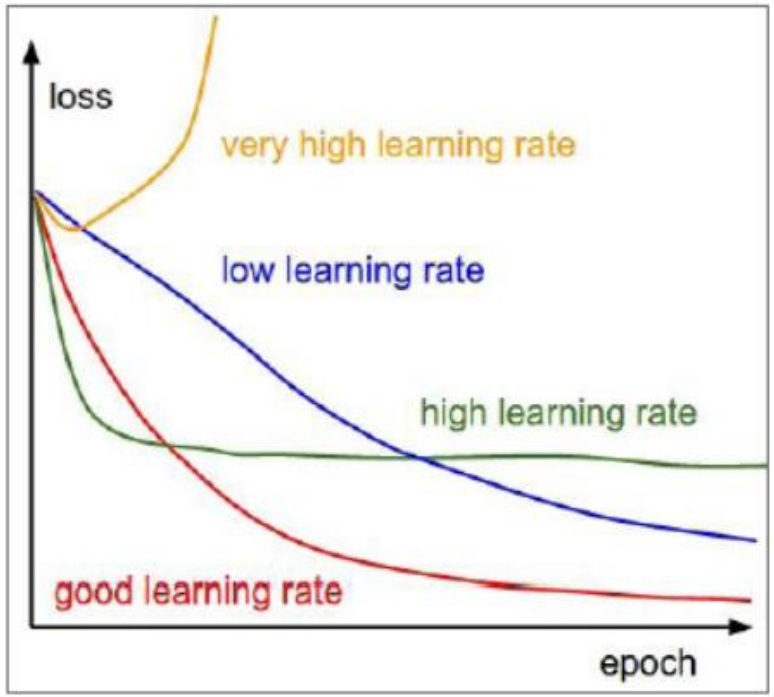
In practice:

- **Adam** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
  - Try cosine schedule, very few hyperparameters!

- If you can afford to do full batch updates then try out **L-BFGS** (and don't forget to disable all sources of noise)

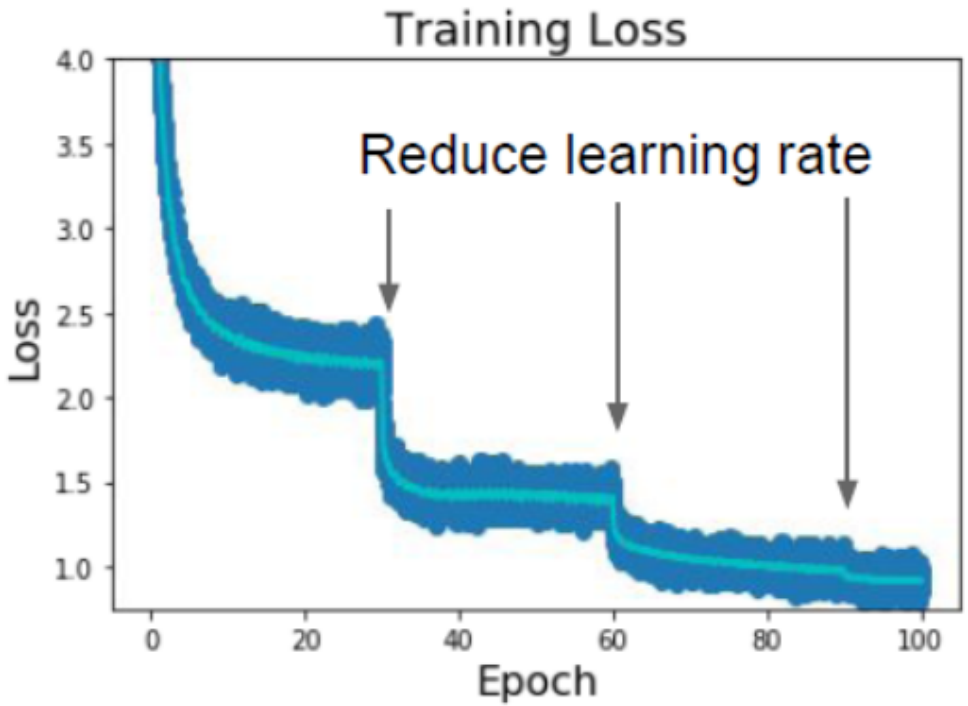
LR: learning rate tuning

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Learning Rate Decay



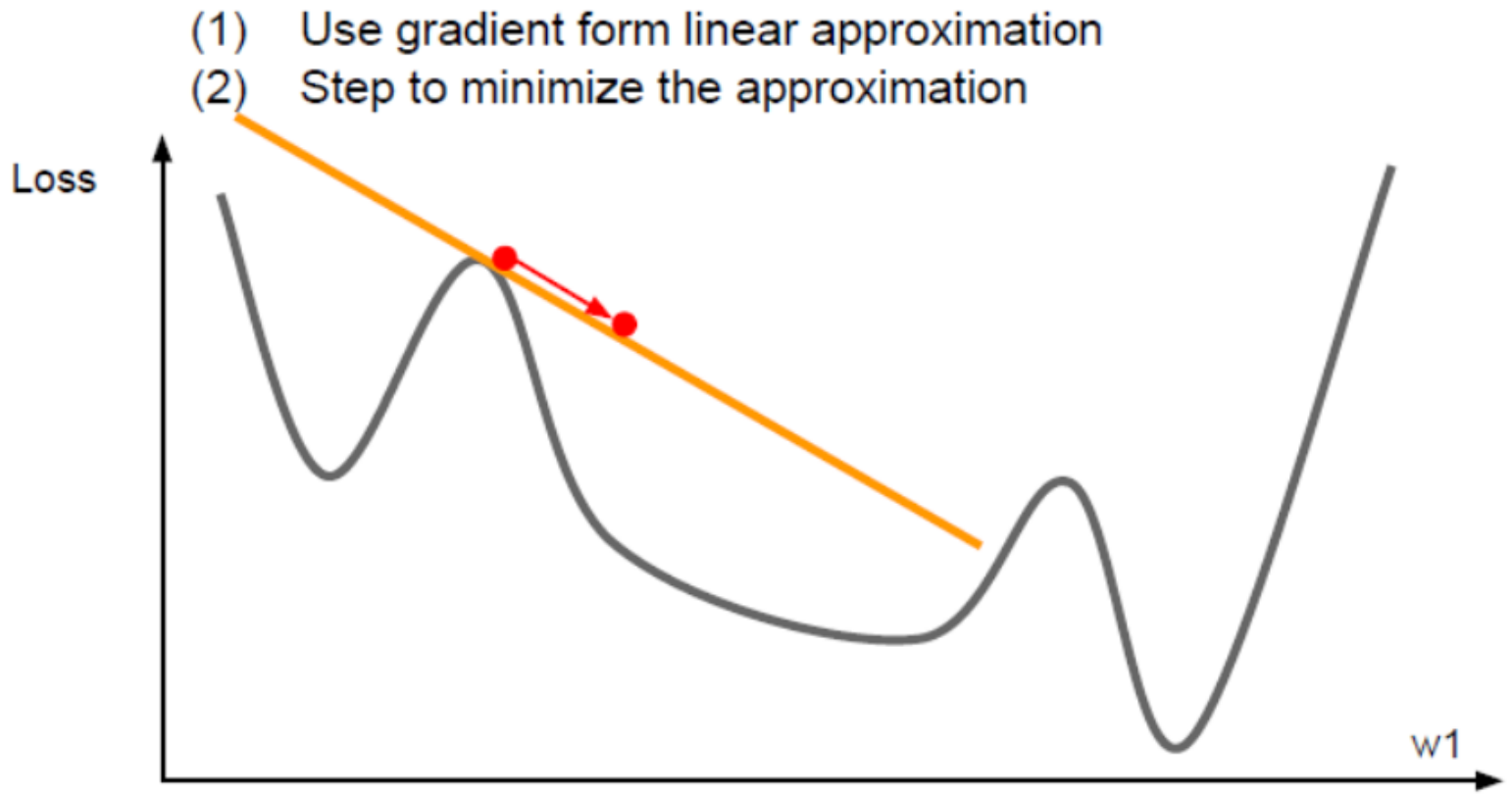
Step: Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

See the animation!!

- http://colah.github.io/posts/2015-08-Understanding-LSTMs/
- https://stackoverflow.com/questions/44273249/in-keras-what-exactly-am-i-configuring-when-i-create-a-stateful-lstm-layer-wi
- http://www.deeplearningbook.org/

Lee's

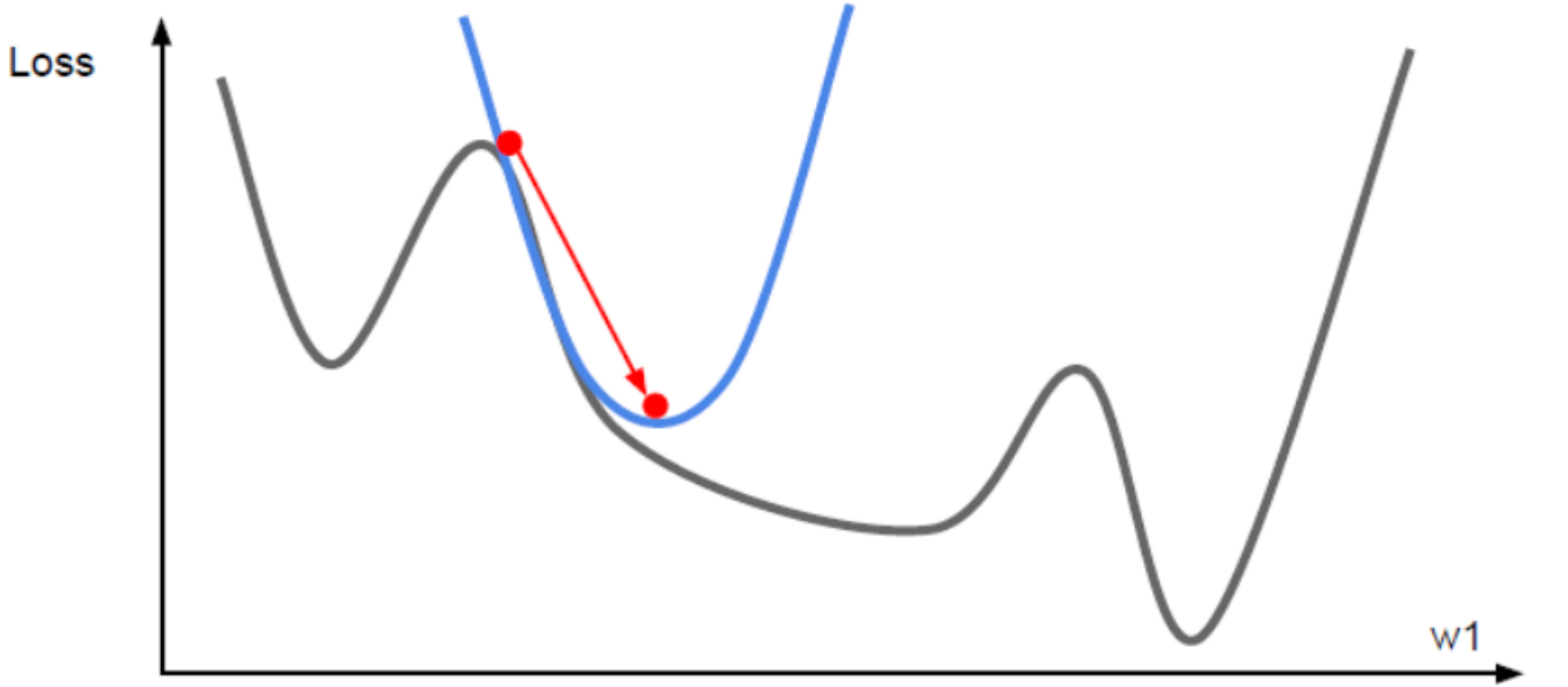
First-Order Optimization



Lee's

Second-Order Optimization

(1) Use gradient **and Hessian** to form **quadratic** approximation
(2) Step to the **minima** of the approximation



Note

- The first order method considers the tangent of a target point, while the second order method considers the curvature of the function near the target point

Second order methods

- Single variable Taylor's Expansion
- $f(x) \cong f(x_0) + (x - x_0)f'(x_0) + \frac{(x-x_0)^2}{2} f''(x_0)$+ …
- Take directive of the equation and let it equals 0.
- $f'(x) = f'(x_0) + (x - x_0)f''(x_0) = 0$
- $x = x_0 - \frac{f'(x_0)}{f''(x_0)}$ …… Newton's method for optimization

- Algorithm Newton's method

  1: let x0 be the initial point

  2: **while**| (f'(x0)| > ϵ **do**

  3: $x = x0 - f'(x0)/f''(x0)$

  4: $x0 = x$

  5: **end while**

  6: **return** x

Multivariate Taylor's expansion

$$J(\theta) \cong J(\theta_0) + (\theta - \theta_0)^t \nabla_\theta J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^t H(\theta - \theta_0)$$

Take directive and equate to 0, we can find

$$\theta^* = \theta_0 - H^{-1}\nabla_\theta J(\theta_0)$$

Where d-dimensional Hessian matrix:

$$H(x) = \begin{bmatrix} \frac{\partial^2 f}{\partial_{x1}\partial_{x1}} & \frac{\partial^2 f}{\partial_{x1}\partial_{x2}} & \cdots \frac{\partial^2 f}{\partial_{x1}\partial_{xd}} \\ \vdots & \vdots & \vdots \\ \frac{\partial^2 f}{\partial_{xd}\partial_{x1}} & \frac{\partial^2 f}{\partial_{xd}\partial_{x2}} & \cdots \frac{\partial^2 f}{\partial_{xd}\partial_{xd}} \end{bmatrix}$$

From Deep learning book

Algorithm 8.8 Newton's method with objective $J(\boldsymbol{\theta}) = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$

Require: Initial parameter $\boldsymbol{\theta}_0$

Require: Training set of m examples

while stopping criterion not met **do**

Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m}\nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute Hessian: $\boldsymbol{H} \leftarrow \frac{1}{m}\nabla^2_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Compute Hessian inverse: $\boldsymbol{H}^{-1}$

Compute update: $\Delta\boldsymbol{\theta} = -\boldsymbol{H}^{-1}\boldsymbol{g}$

Apply update: $\boldsymbol{\theta} = \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$

end while

Lee's

Second-Order Optimization

second-order Taylor expansion:

$$J(\boldsymbol{\theta}) \approx J(\boldsymbol{\theta}_0) + (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0) + \frac{1}{2}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top \boldsymbol{H}(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

Solving for the critical point we obtain the Newton parameter update:

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

Hessian has O(N^2) elements
Inverting takes O(N^3)
N = (Tens or Hundreds of) Millions

Q: Why is this bad for deep learning?

Lee's

Second-Order Optimization

$$\boldsymbol{\theta}^* = \boldsymbol{\theta}_0 - \boldsymbol{H}^{-1} \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_0)$$

- Quasi-Newton methods (**BGFS** most popular):
  instead of inverting the Hessian (O(n^3)), approximate inverse Hessian with rank 1 updates over time (O(n^2) each).

- **L-BFGS** (Limited memory BFGS):
  Does not form/store the full inverse Hessian.

Lee's

L-BFGS

- **Usually works very well in full batch, deterministic mode** i.e. if you have a single, deterministic f(x) then L-BFGS will probably work very nicely

- **Does not transfer very well to mini-batch setting**. Gives bad results. Adapting second-order methods to large-scale, stochastic setting is an active area of research.

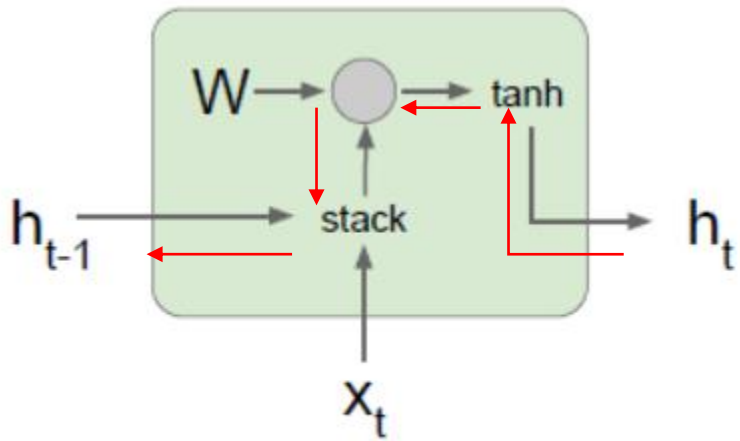
Gradient vanishing & exploding in RNN

- Recap

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013

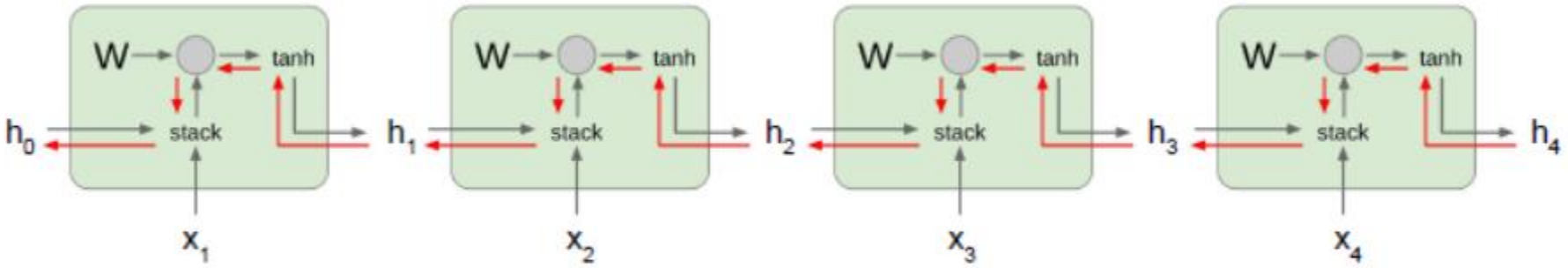
Backpropagation from h_t to h_{t-1} multiplies by W (actually W_{hh}^T)



$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$
$$= \tanh\left((W_{hh} \quad W_{hx})\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$
$$= \tanh\left(W\begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}\right)$$

Vanilla RNN Gradient Flow

Bengio et al, "Learning long-term dependencies with gradient descent is difficult", IEEE Transactions on Neural Networks, 1994
Pascanu et al, "On the difficulty of training recurrent neural networks", ICML 2013



Computing gradient of h_0 involves many factors of W (and repeated tanh)

Consider only W_{hh} , $h_t = W_{hh} . h_{t-1}$

- **Let** ξ be the cost function: the gradient of m steps before the current step t is:
-

$$\frac{\partial \xi}{\partial h_{t-m}} = \frac{\partial \xi}{\partial h_t} * \frac{\partial h_t}{\partial h_{t-m}}$$

Where $\frac{\partial h_t}{\partial h_{t-m}} = \frac{\partial h_t}{\partial h_{t-1}} * \cdots * \frac{\partial h_{t-m+1}}{\partial h_{t-m}} = W^m$

$$\frac{\partial \xi}{\partial h_{t-m}} = \frac{\partial \xi}{\partial h_t} * W^m$$

Deep learning book

- Suppose W has an eigendecomposition $\boldsymbol{W}=\boldsymbol{V}\,diag(\boldsymbol{\lambda})\boldsymbol{V}^{-1}$
- $\boldsymbol{W}^{m} = (\boldsymbol{V}diag(\boldsymbol{\lambda})\boldsymbol{V}^{-1})^{m}=\boldsymbol{V}\,diag(\boldsymbol{\lambda})^{m}\boldsymbol{V}^{-1}$
- Any eigenvalues λ_i that are not near an absolute value of 1 will either explode if they are greater than 1 in magnitude or vanish if they are less than 1 in magnitude.
- Vanishing gradients make it difficult to know which direction the parameters should move to improve the cost function
- Exploding gradients can make learning unstable

- Recurrent networks use **the same matrix W** at each time step, but **feedforward networks do not**, so even very deep feedforward networks can largely avoid the vanishing and exploding gradient problem.

- ---from deep learning book of Ian Goodfellow etc.

A diagonalization example

- $A = \begin{bmatrix} -4 & -5 \\ 10 & 11 \end{bmatrix}$ *has eigen value* $\lambda_1 = 1,\ x_1 = \begin{bmatrix} 1 \\ -1 \end{bmatrix}$,

$$\lambda_2 = 6, \qquad x_2 = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$$

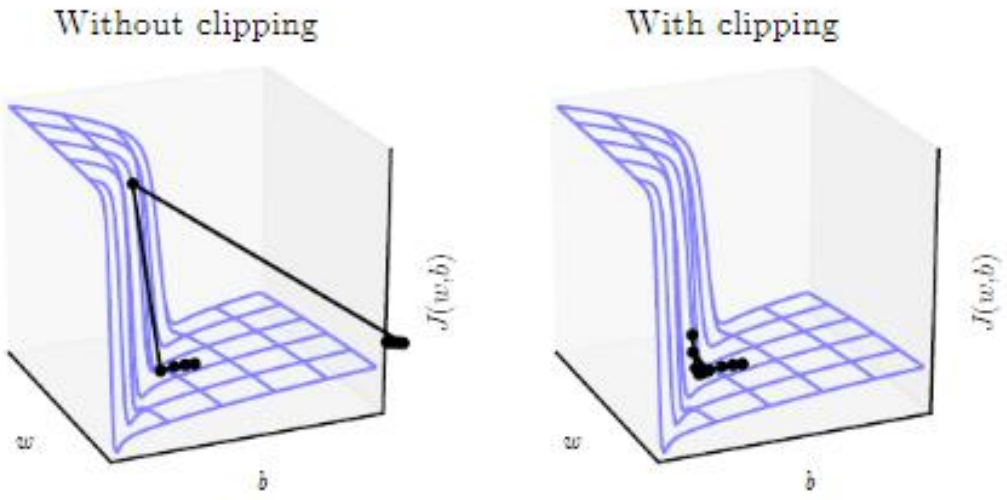
$$A^k = (S\Lambda S^{-1})^k = S\Lambda^k S^{-1} = \begin{bmatrix} 1 & -1 \\ -1 & 2 \end{bmatrix} \begin{bmatrix} 1^k & 0 \\ 0 & 6^k \end{bmatrix} \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 2 - 6^k & 1 - 6^k \\ -2 + 2 \cdot 6^k & -1 + 2 \cdot 6^k \end{bmatrix}$$

To deal with the gradient exploding problem

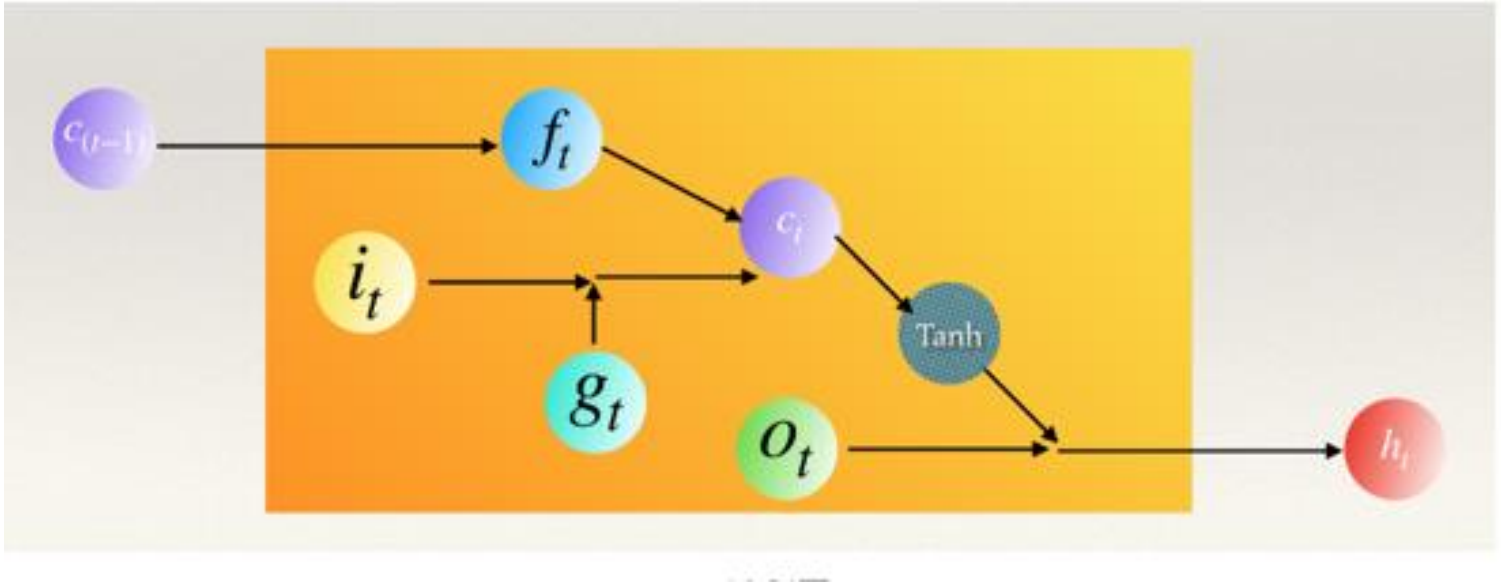
1. Gradient clipping, where we threshold the maximum value a gradient can get

$$if\ \|\boldsymbol{g}\| > \beta,\ g \leftarrow \frac{\beta\ \boldsymbol{g}}{\|\boldsymbol{g}\|}$$



Training LSTM

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t c_{(t-1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

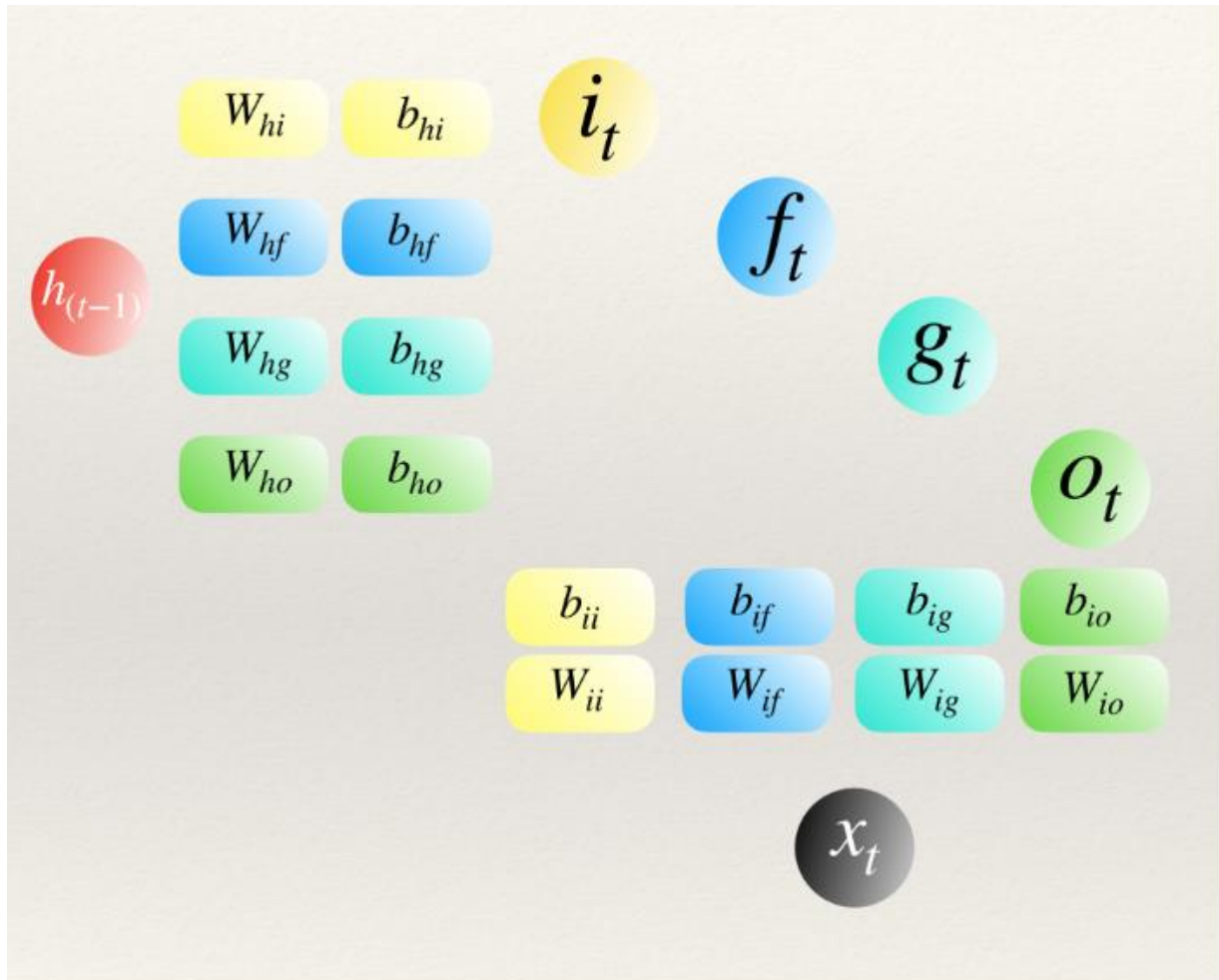


f: forget gate

i: input gate

g: gate gate (or cell gate)

o: output gate


W_{hi}
b_{hi}
i_t
W_{hf}
b_{hf}
f_t
$h_{(t-1)}$
W_{hg}
b_{hg}
g_t
W_{ho}
b_{ho}
o_t
b_{ii}
b_{if}
b_{ig}
b_{io}
W_{ii}
W_{if}
W_{ig}
W_{io}
x_t

To simplify, set x_t and h_t

$$x_t = [x_1, x_2, \ldots, x_5]$$

輸入之向量

$$h_{(t-1)} = [h_1, h_2, \ldots, h_{10}]$$

上一刻的輸出

$$W_{ii} = [W_{ii1}, \ldots\ldots, W_{ii10}] = \begin{bmatrix} W_{ii1,1}, \ldots, W_{ii10,1} \\ \cdots \\ W_{ii1,5}, \ldots, W_{ii10,5} \end{bmatrix}$$

W_{ii} is 5*10; W_{hi} is 10*10

$$W_{hi} = [W_{hi1} \cdots W_{hi10}]$$
$$= \begin{bmatrix} W_{hi1,1} \cdots W_{hi10,1} \\ W_{hi1,2} \cdots W_{hi10,2} \\ \cdots\cdots \\ W_{hi1,10} \cdots W_{hi10,10} \end{bmatrix}$$

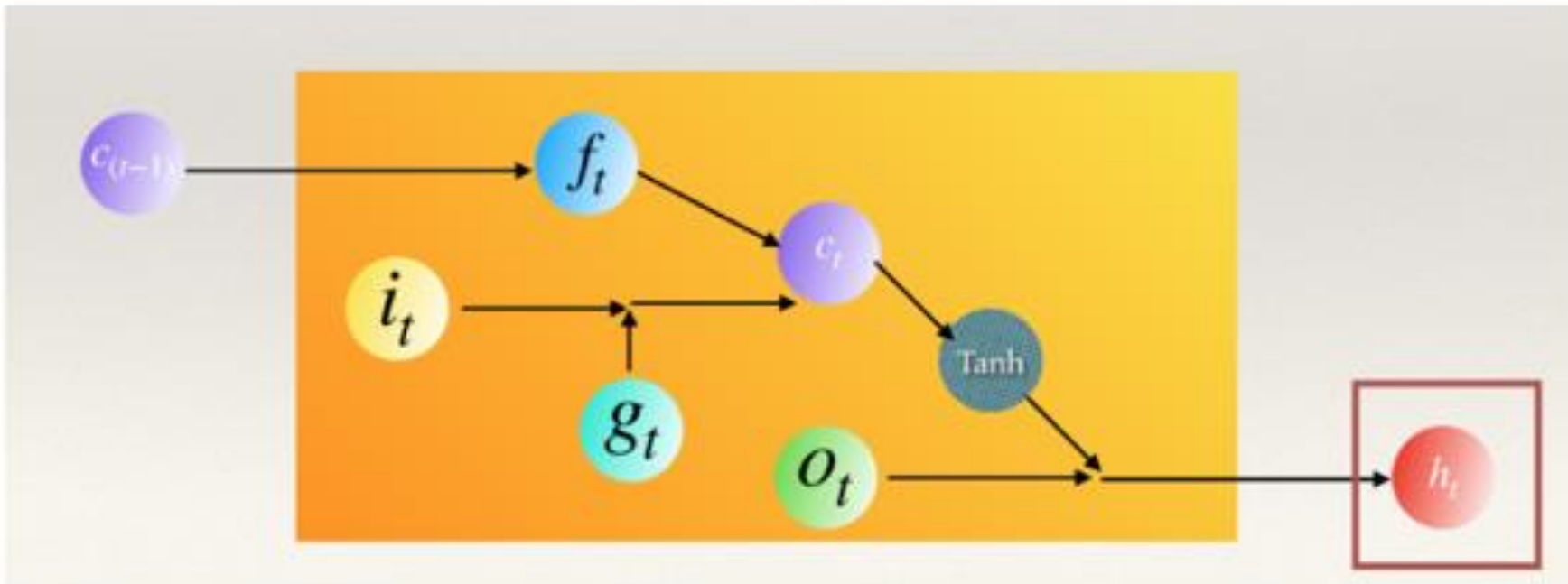
X is 1*5 matrix, weight W_{ii} is 5*10 matrix to generate a vector of 1*10
To be added to the hidden state
Let ignore the bias.

$$y_i = h_{(t-1)}W_{hi} + x_tW_{ii} + b_{ii} + b_{hi}$$

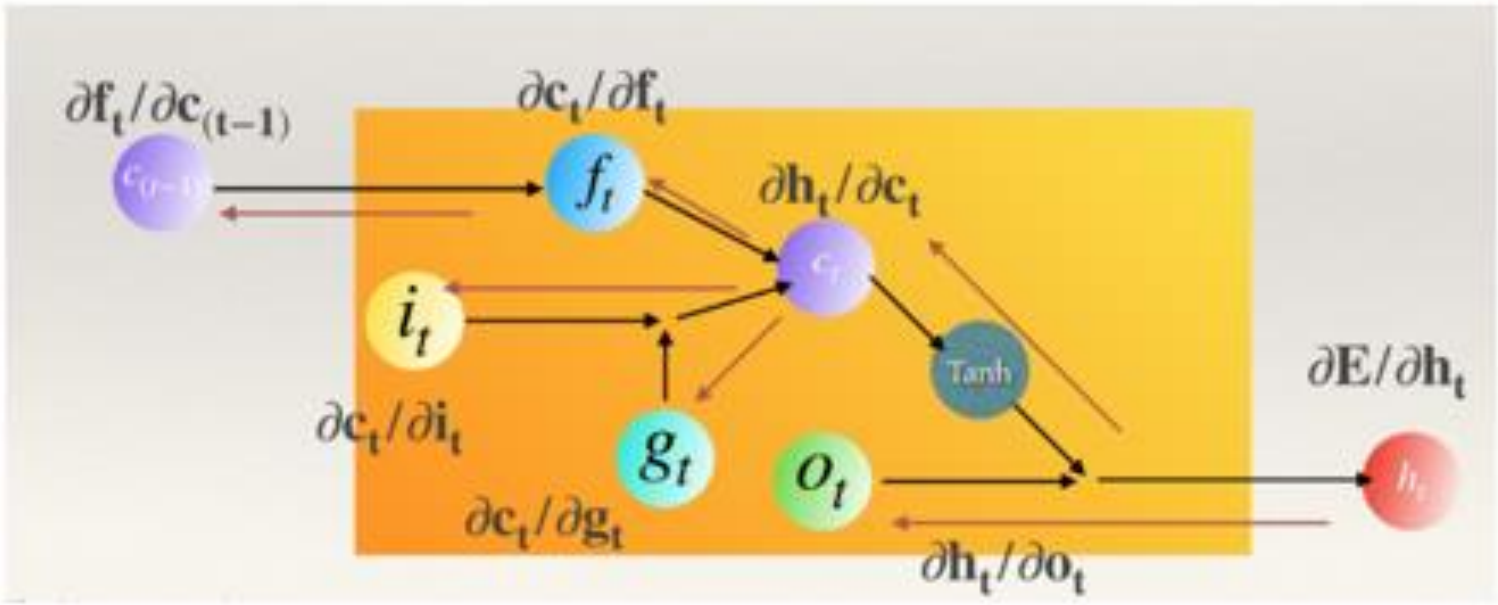
$$i_t = \sigma(y_i) = [i_{t1}, \ldots, i_{t10}]$$

Element-wise operation

$$i_t = \sigma(y_i) = [i_{t1}, \ldots, i_{t10}]$$
$$f_t = \sigma(y_f) = [f_{t1}, \ldots, f_{t10}]$$
$$g_t = \tanh(y_g) = [g_{t1}, \ldots, g_{t10}]$$
$$o_t = \sigma(y_o) = [o_{t1}, \ldots, o_{t10}]$$


$c_{(t-1)}$
f_t
c_t
i_t
Tanh
g_t
o_t
h_t

LSTM backward propagation (forget gate)



Chain rule

$$\partial \mathbf{E}/\partial \mathbf{f}_t = (\partial \mathbf{E}/\partial \mathbf{h}_t)(\partial \mathbf{h}_t/\partial \mathbf{c}_t)(\partial \mathbf{c}_t/\partial \mathbf{f}_t)$$

$$\partial \mathbf{h}_t/\partial \mathbf{c}_t = \mathbf{o}_t(1 - \tanh^2(\mathbf{c}_t))$$

$$\partial \mathbf{c}_t/\partial \mathbf{f}_t = \mathbf{c}_{(t-1)}$$

Training LSTM

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}) \\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}) \\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}) \\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}) \\
c_t &= f_t c_{(t-1)} + i_t g_t \\
h_t &= o_t \tanh(c_t)
\end{aligned}
$$

$$\partial \mathbf{f_t}/\partial \mathbf{W_{if}} = \begin{bmatrix} \partial f_{t1}/\partial W_{fi1,1}, \ldots, \partial f_{t10}/\partial W_{fi10,1} \\ \cdots \\ \partial f_{t1}/\partial W_{fi1,5}, \ldots, \partial f_{t10}/\partial W_{fi10,5} \end{bmatrix}$$

where $\partial f_{ti}/\partial W_{fii,j} = f_{ti}(1 - f_{ti})x_{ti}$

Put them together

$$\partial \mathbf{E}/\partial \mathbf{W}_{\mathbf{if}} = (\partial \mathbf{E}/\partial \mathbf{h}_t)(\partial \mathbf{h}_t/\partial \mathbf{c}_t)(\partial \mathbf{c}_t/\partial \mathbf{f}_t)(\partial \mathbf{f}_t/\partial \mathbf{W}_{\mathbf{if}})$$

$$\partial \mathbf{h}_t/\partial \mathbf{c}_t = \mathbf{o}_t(1 - \tanh^2(\mathbf{c}_t))$$

$$\partial \mathbf{c}_t/\partial \mathbf{f}_t = \mathbf{c}_{(t-1)}$$

$$\partial \mathbf{f}_t/\partial \mathbf{W}_{\mathbf{if}} = \begin{bmatrix} \partial f_{t1}/\partial W_{fi1,1}, \ldots, \partial f_{t10}/\partial W_{fi10,1} \\ \cdots \\ \partial f_{t1}/\partial W_{fi1,5}, \ldots, \partial f_{t10}/\partial W_{fi10,5} \end{bmatrix}$$

where $\partial f_{ti}/\partial W_{fii,j} = f_{ti}(1 - f_{ti})x_{ti}$

$$\therefore \ \mathbf{W}_{\mathbf{if,new}} = \mathbf{W}_{\mathbf{if}} - \gamma(\partial \mathbf{E}/\partial \mathbf{W}_{\mathbf{if}})$$

Reference

1. https://www.bioing.jku.at/publications/older/2604.pdf
2. https://pytorch.org/docs/stable/nn.html?hight=lstm#torch.nn.LSTM

Number of parameters

- Hidden state weight matrix has (hidden state length)^2 parameters
- Input matrix has (hidden state length) * input vector length
- We have 4 sets of gate's matrices to train
- Many parameters to train