Neural Networks 401

Deep Learning for Natural Language Processing

Vladislav Lialin, Text Machine Lab

Administrative

- Python Code style quiz grades are released
  - Next step: working on your quiz mistakes
  - Complete this google form before the next class
    https://forms.gle/VB8Lmyxv11bUfS1C9
- Homework 3 is extended until the next class
  - If you already submitted, take a second look and fix your mistakes
  - Make sure you **understand** it, you will need to do similar things on your final exam
  - Extra points for implementing new activation functions.
    Try GELU, Swish, invent your own.
    Make sure to check the gradients numerically using `eval_numerical_gradient_array()`

What you should remember after this lecture

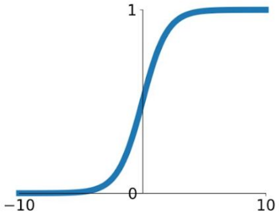
- Momentum and ADAM optimizers
- Batch Normalization
- Dropout
- Neural networks are hard to train

Neural Networks 301
Recap

Activation functions

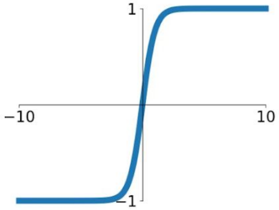
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



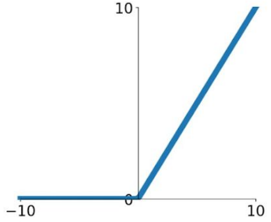
tanh

$$\tanh(x)$$



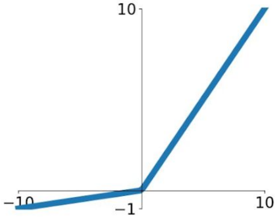
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

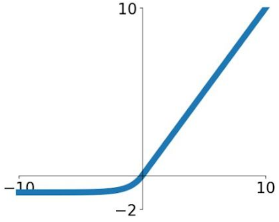


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$

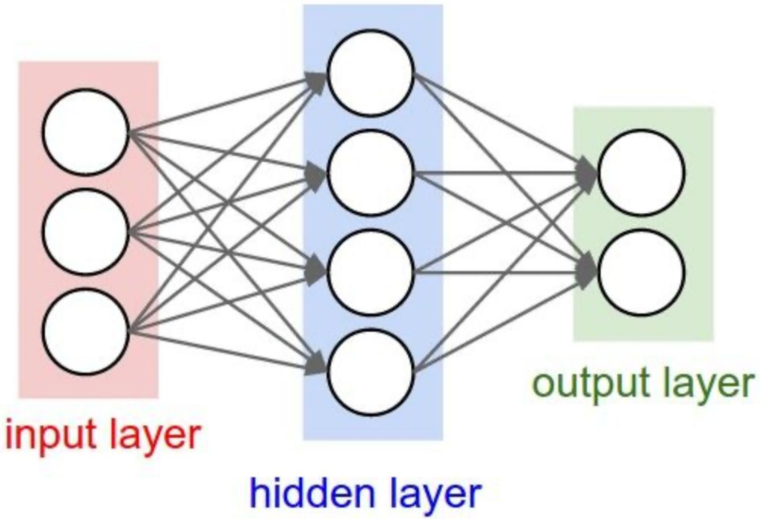


Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Weight initialization

- Q: what happens when W=constant init is used?

$$y_1 = xW_1 + b_1$$
$$y_2 = y_1 W_2 + b_2$$
$$l = l(y_2, y_{target})$$
$$\frac{dl}{dW_2^{i,j}} = \frac{dl}{dy_2^j} \cdot \frac{dy_2^j}{dW_2^{i,j}}$$



If all Wij are the same, then gradients will be **different** for Wij across **i**-dimension,
but **exactly the same** across **j**-dimension — hidden dimension.
So effectively hidden dimension is 1 and every **hidden neuron acts exactly the same**.

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

First idea: Small random numbers

```
W = 0.01 * np.random.randn(Din, Dout)
```

Works well for small networks, requires tuning the range for large networks

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Activation statistics, tanh nonlinearity

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

What happens to the activations of the first layer?

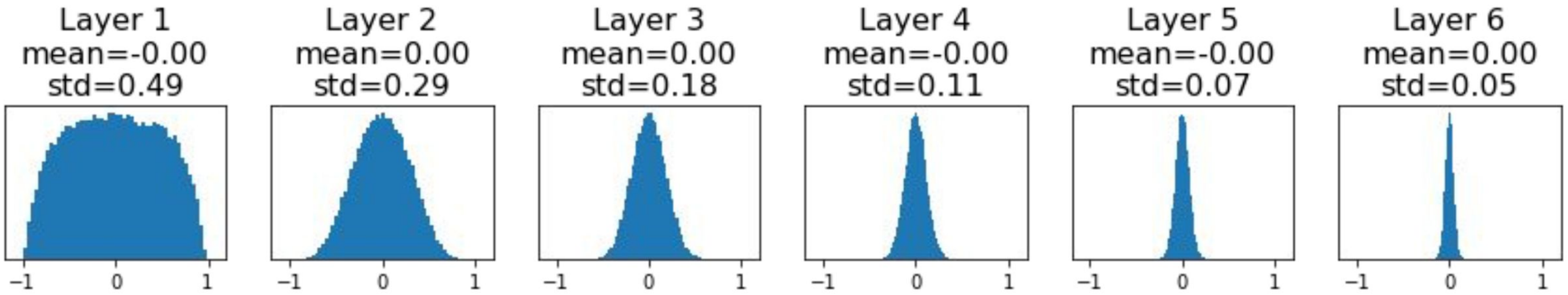
Activation statistics, tanh nonlinearity

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Vanishing gradients

Activation statistics, tanh nonlinearity

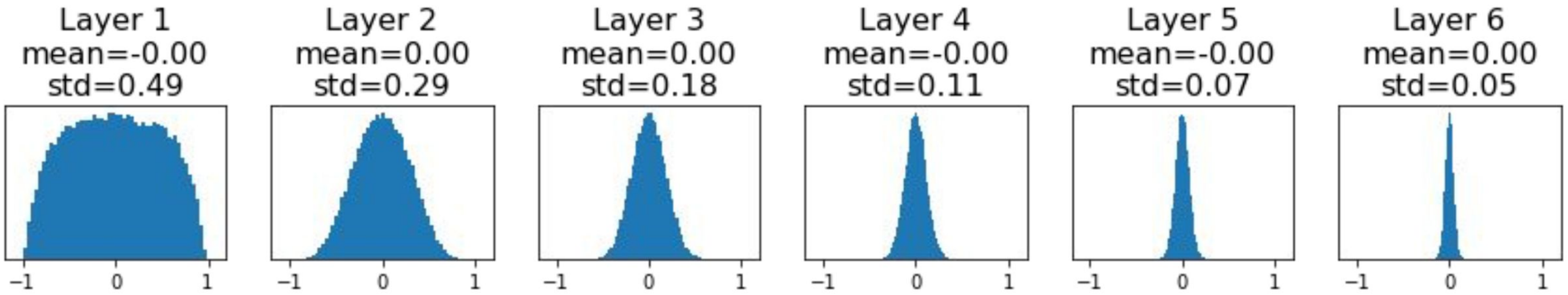
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.01 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Forward pass for a 6-layer net with hidden size 4096

All activations tend to zero for deeper network layers

Q: What do the gradients dL/dW look like?

A: Zero gradients, no learning



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

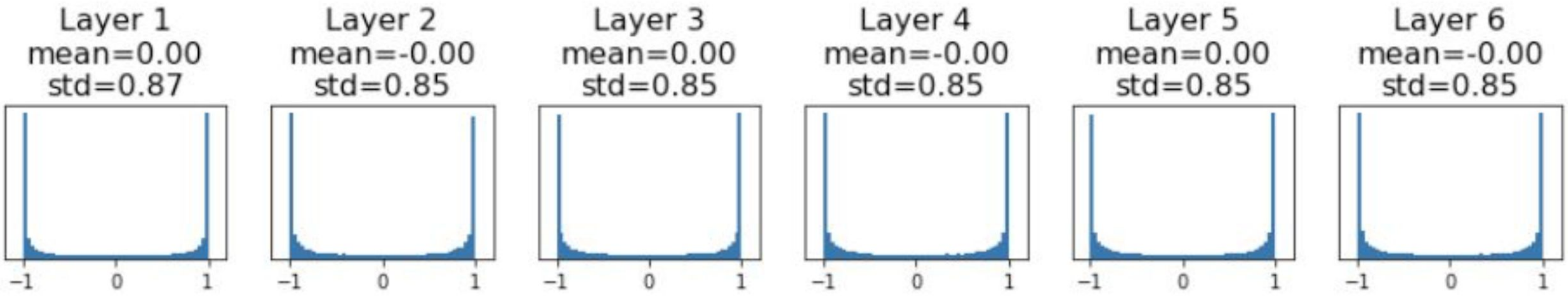
Activation statistics, tanh nonlinearity

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

Activation statistics, tanh nonlinearity

Vanishing gradients

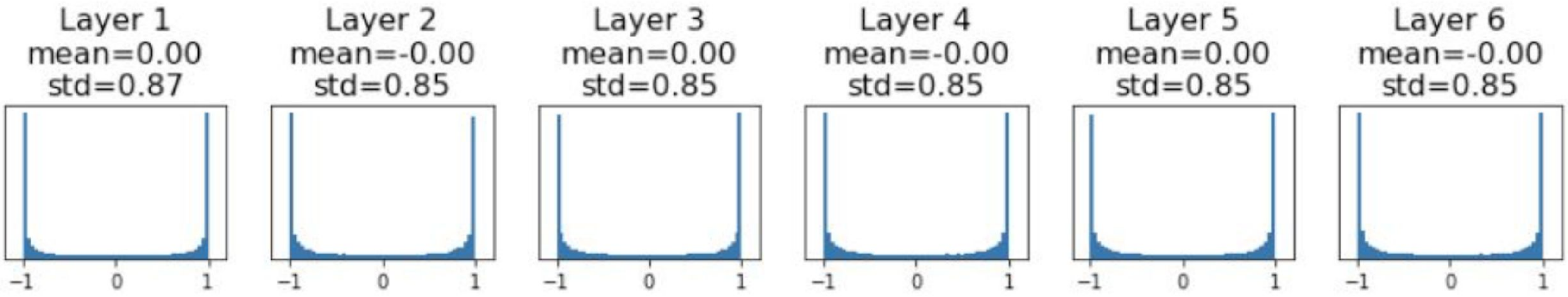
```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = 0.05 * np.random.randn(Din, Dout)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

Increase std of initial weights from 0.01 to 0.05

All activations saturate

Q: What do the gradients look like?

A: Zero local gradients, no learning



Slide credit: Stanford CS231n
Image credit: Stanford CS231n

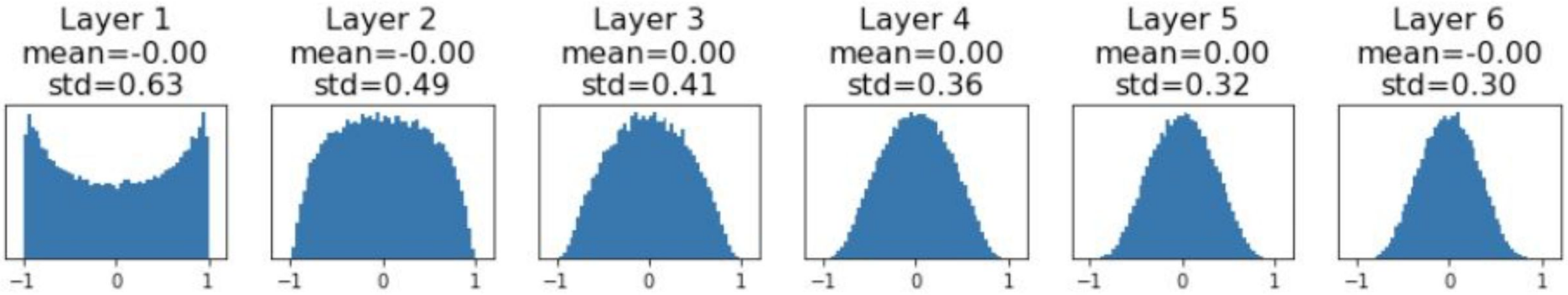
We want the initialization range to be not too big and not too small

Xavier Initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.tanh(x.dot(W))
    hs.append(x)
```

"Xavier" initialization:
std = 1/sqrt(Din)

"Just right": Activations are nicely scaled for all layers!



Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Understanding the difficulty of training deep feedforward neural networks, Glorot and Bengio, 2010

Xavier Initialization

$y = Wx$
$h = f(y)$

Derivation:

$$\begin{aligned} Var(y_i) &= Din * Var(x_i w_i) && \text{[Assume x, w are iid]} \\ &= Din * (E[x_i^2]E[w_i^2] - E[x_i]^2\,E[w_i]^2) && \text{[Assume x, w independant]} \\ &= Din * Var(x_i) * Var(w_i) && \text{[Assume x, w are zero-mean]} \end{aligned}$$

If $Var(w_i) = 1/Din$ then $Var(y_i) = Var(x_i)$

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Understanding the difficulty of training deep feedforward neural networks, Glorot and Bengio, 2010

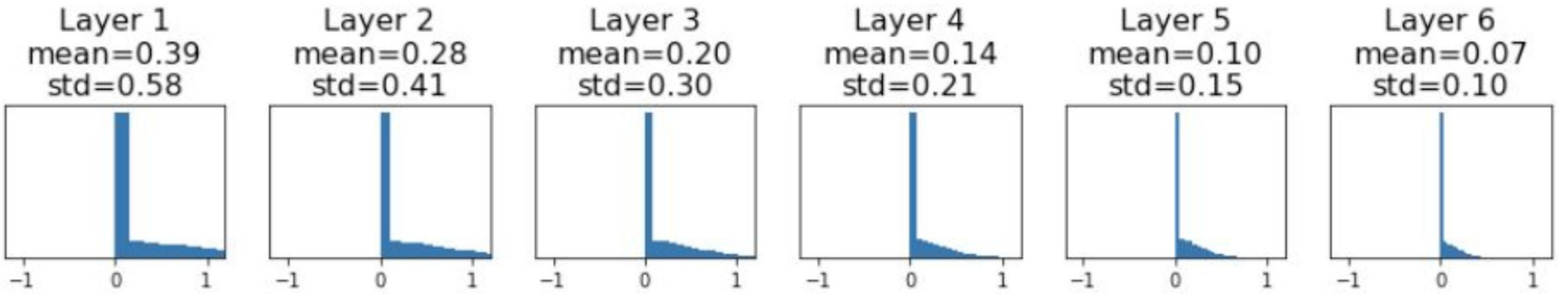
Xavier Initialization with ReLU

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) / np.sqrt(Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

Change from tanh to ReLU

Xavier assumes zero centered activation function

Activations collapse to zero again, no learning =(



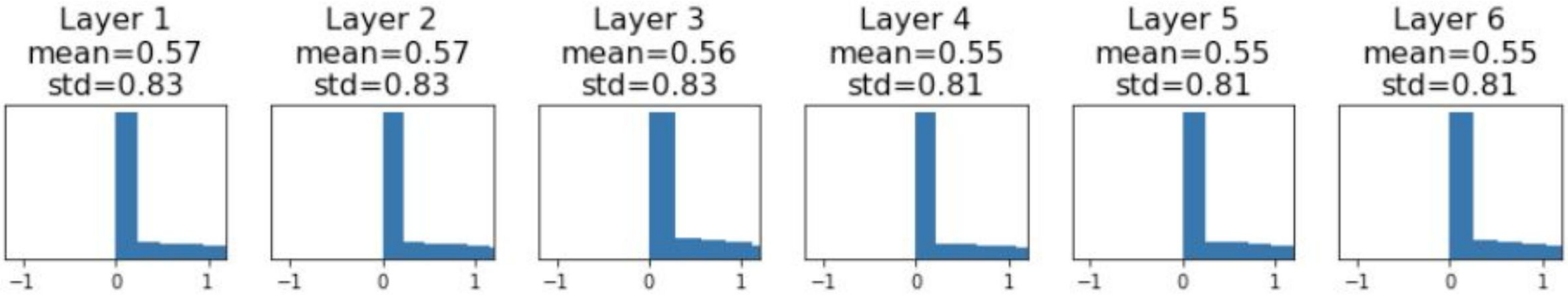
Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He et al., 2016

Kaiming initialization

```
dims = [4096] * 7
hs = []
x = np.random.randn(16, dims[0])
for Din, Dout in zip(dims[:-1], dims[1:]):
    W = np.random.randn(Din, Dout) * np.sqrt(2/Din)
    x = np.maximum(0, x.dot(W))
    hs.append(x)
```

ReLU correction: std = sqrt(2 / Din)

"Just right": Activations are nicely scaled for all layers!



Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, He et al., 2016

Practical tips

- **Initialization matters**
- Large init is bad
- Small init is bad
- Xavier and Kaiming provide a good heuristic to get “just right” init range
- For very large networks (> 100M parameters) normal distribution produces too many outliers (very large numbers) which can heart convergence
    - Using uniform distribution instead of normal helps

$$XavierNormal(d_{in}) = \sqrt{\frac{1}{d_{in}}} N(0,1) \qquad XavierUniform(d_{in}) = \sqrt{\frac{6}{d_{in}}} U(0,1)$$

$$KaimingNormal(d_{in}) = \sqrt{\frac{2}{d_{in}}} N(0,1) \qquad KaimingUniform(d_{in}) = \sqrt{\frac{3}{d_{in}}} N(0,1)$$

Optimization

Minibatch training

- True gradient computation requires to process full dataset
    - Full dataset Just to do a single weights update!

Minibatch training

- True gradient computation requires to process full dataset
  - Full dataset Just to do a single weights update!
- Solution: minibatch training
  - Sample N examples from your dataset
  - Perform forward() and backward(), average results
  - Get (noisy) stochastic gradient
  - Update your parameters

Minibatch training

- True gradient computation requires to process full dataset
  - Full dataset Just to do a single weights update!
- Solution: minibatch training
  - Sample N examples from your dataset
  - Perform forward() and backward(), average results
  - Get (noisy) stochastic gradient
  - Update your parameters
- What is the best N?
  - As big as your GPU memory can fit
  - Try to keep it at least 32
  - Bigger networks (tens of layers) need bigger batches

Minibatch training

- True gradient computation requires to process full dataset
  - Full dataset Just to do a single weights update!
- Solution: minibatch training
  - Sample N examples from your dataset
  - Perform forward() and backward(), average results
  - Get (noisy) stochastic gradient
  - Update your parameters
- What is the best N?
  - As big as your GPU memory can fit
  - Try to keep it at least 32
  - Bigger networks (tens of layers) need bigger batches
- Gradient descent with minibatches is called stochastic gradient descent (SGD)

Epoch training

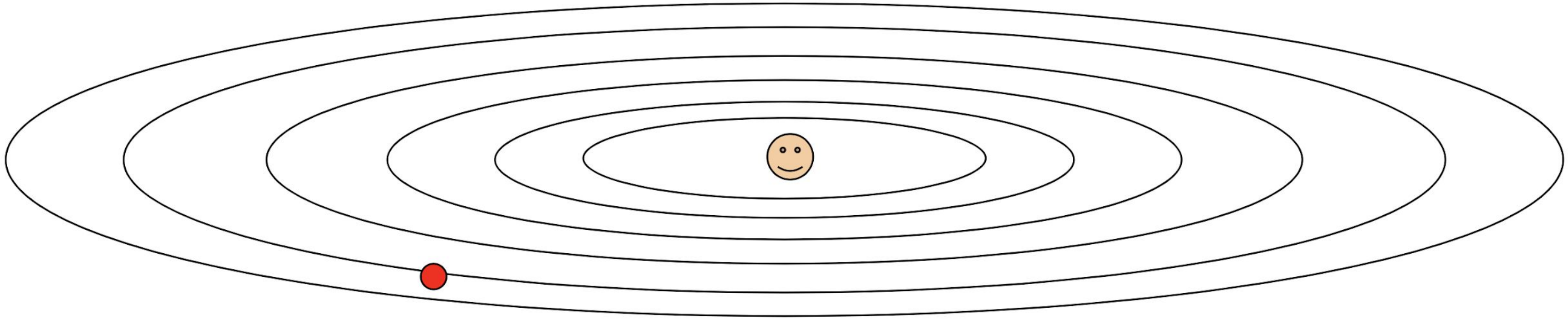
- Uniform sampling from the dataset does not guarantee that every example in the dataset will be used for a finite number of samples

Epoch training

- Uniform sampling from the dataset does not guarantee that every example in the dataset will be used for a finite number of samples
- Better approach:
  - Shuffle your dataset
  - For every step take N consequent examples
  - Repeat until the end of the dataset
  - Shuffle the dataset again
- One dataset pass is called an epoch

SGD problems

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

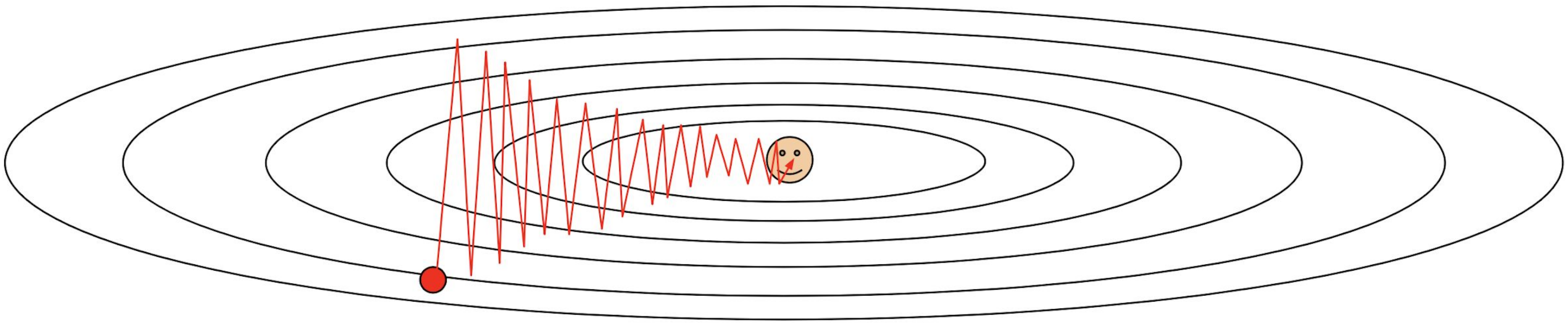


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

SGD problems

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

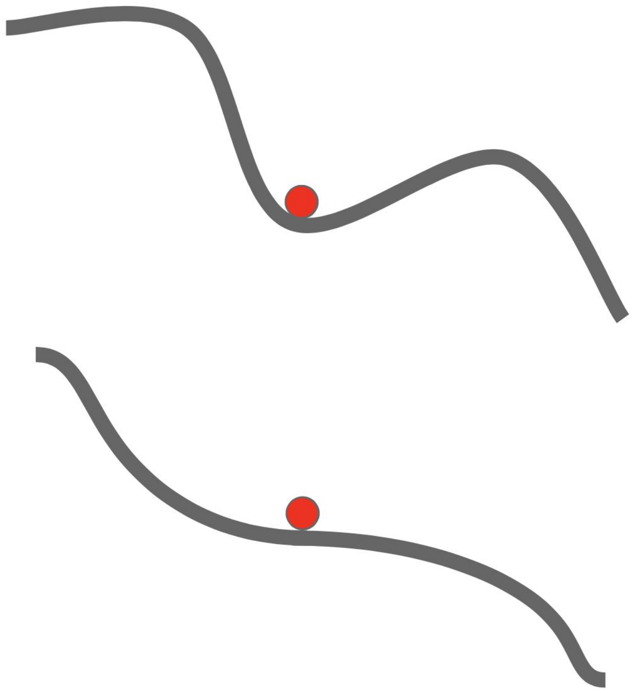


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Slide credit: Stanford CS231n
Image credit: Stanford CS231n

SGD Problems

- Saddle points
  - Gradient is near-zero, but this is not a minimum
  - Saddle points are extremely common
- Local minima
  - Not a problem in practice
  - Reasons to believe that SGD finds a global minimum for neural networks with enough parameters
- Noisy gradients
  - Just use larger batches



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

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
On the importance of initialization and momentum in deep learning, Sutskever et al, 2013

Momentum visualization

[Click]

AdaGrad: 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)
```

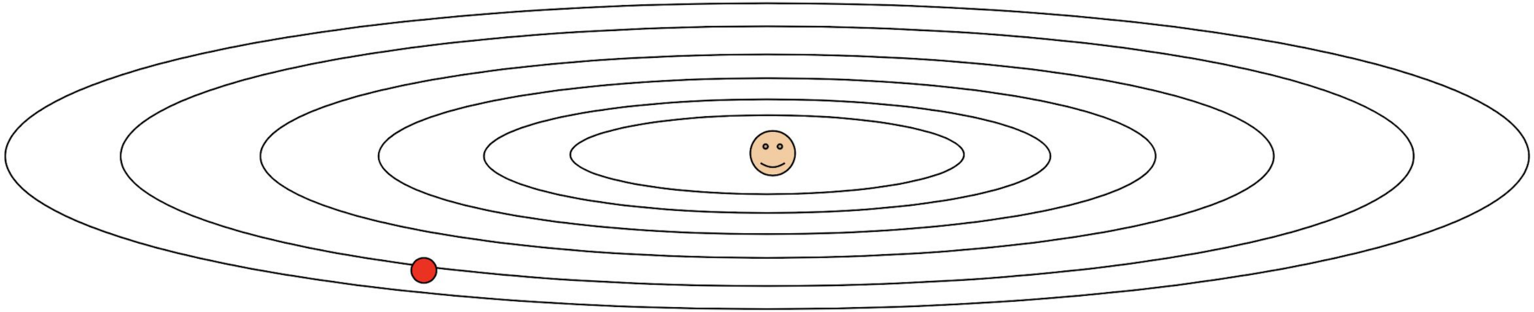
Added element-wise scaling of the gradient based on the historical sum of squares in each dimension

"Per-parameter learning rates" or "adaptive learning rates"

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Adaptive subgradient methods for online learning and stochastic optimization, Duchi et al, 2011

AdaGrad: 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)
```

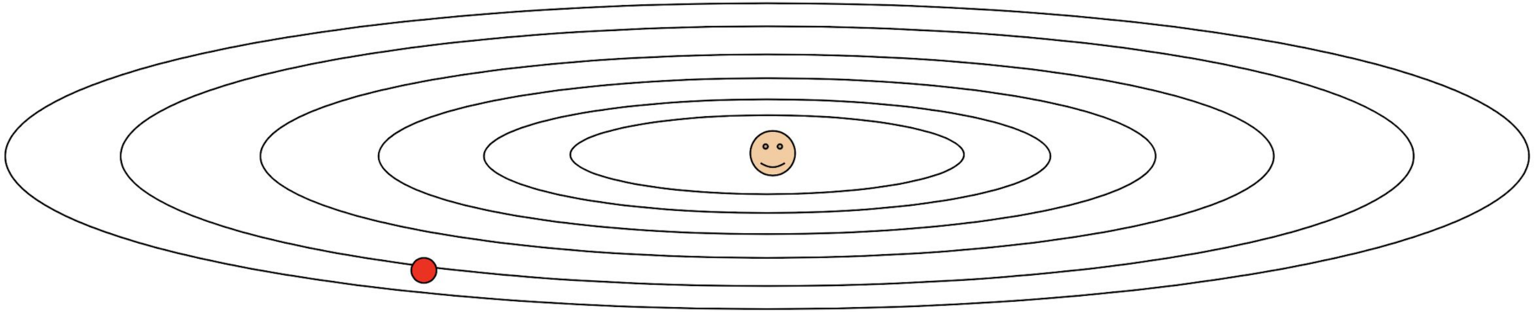


Q: What happens with AdaGrad?

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Adaptive subgradient methods for online learning and stochastic optimization, Duchi et al, 2011

AdaGrad: 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)
```



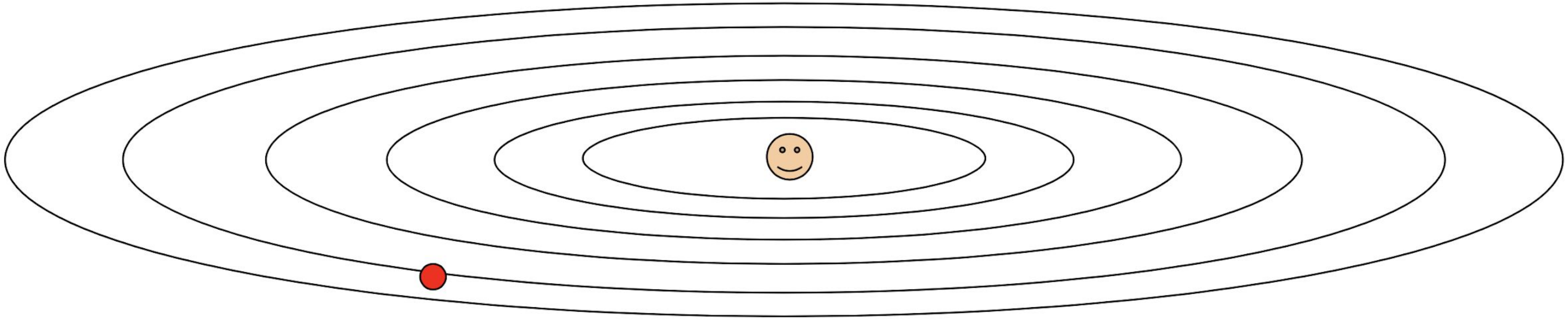
Q: What happens with AdaGrad?

A: Progress along “steep” dimensions is dumped
Progress along “flat” dimensions is accelerated

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Adaptive subgradient methods for online learning and stochastic optimization, Duchi et al, 2011

AdaGrad: 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)
```

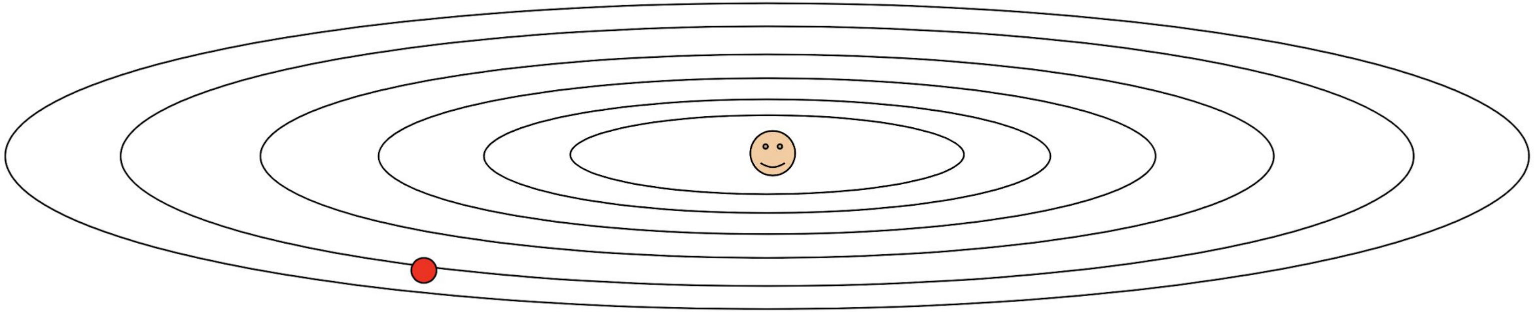


Q2: What happens to the step size over long time?

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Adaptive subgradient methods for online learning and stochastic optimization, Duchi et al, 2011

AdaGrad: 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)
```



Q: What happens with AdaGrad?

A: gradient decays to zero

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Adaptive subgradient methods for online learning and stochastic optimization, Duchi et al, 2011

RMSProp: AdaGrad fix

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)
```

Slide credit: Stanford CS231n
Image credit: Stanford CS231n
Tieleman and Hinton, 2012

Momentum + RMSProp = Adam

```
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

Sort of like RMSProp with momentum

Q: What happens at first timestep?

ADAM issues

- Consumes three times the memory of your network
- Biased updates with L2 regularization
  - Solution: AdamW
- Needs “warmup” at first steps to “learn” the loss landscape if you want to achieve the best results
- Needs learning rate tuning (extremely important)
- Modern alternatives:
  - Adafactor — ADAM, but the matrix of moments are factorized which sames memory
  - 8-bit ADAM — uses 8-bit numbers instead of 32-bit numbers for the moments
  - Distributed Shampoo — second-order method that works great if you have more than 8 GPUs
  - Lion — Feb 13 2023, **3 days ago**

Optimizers is an active area of research

- Adafactor — ADAM, but the moments are factorized to save memory
- 8-bit ADAM — uses 8-bit numbers instead of 32-bit numbers for the moments
- Distributed Shampoo — second-order method that works great if you have more than 8 GPUs
- Lion — Feb 13 2023, **3 days ago**
  Only considers the sign of the gradient.