Training Aspects of Neural Networks

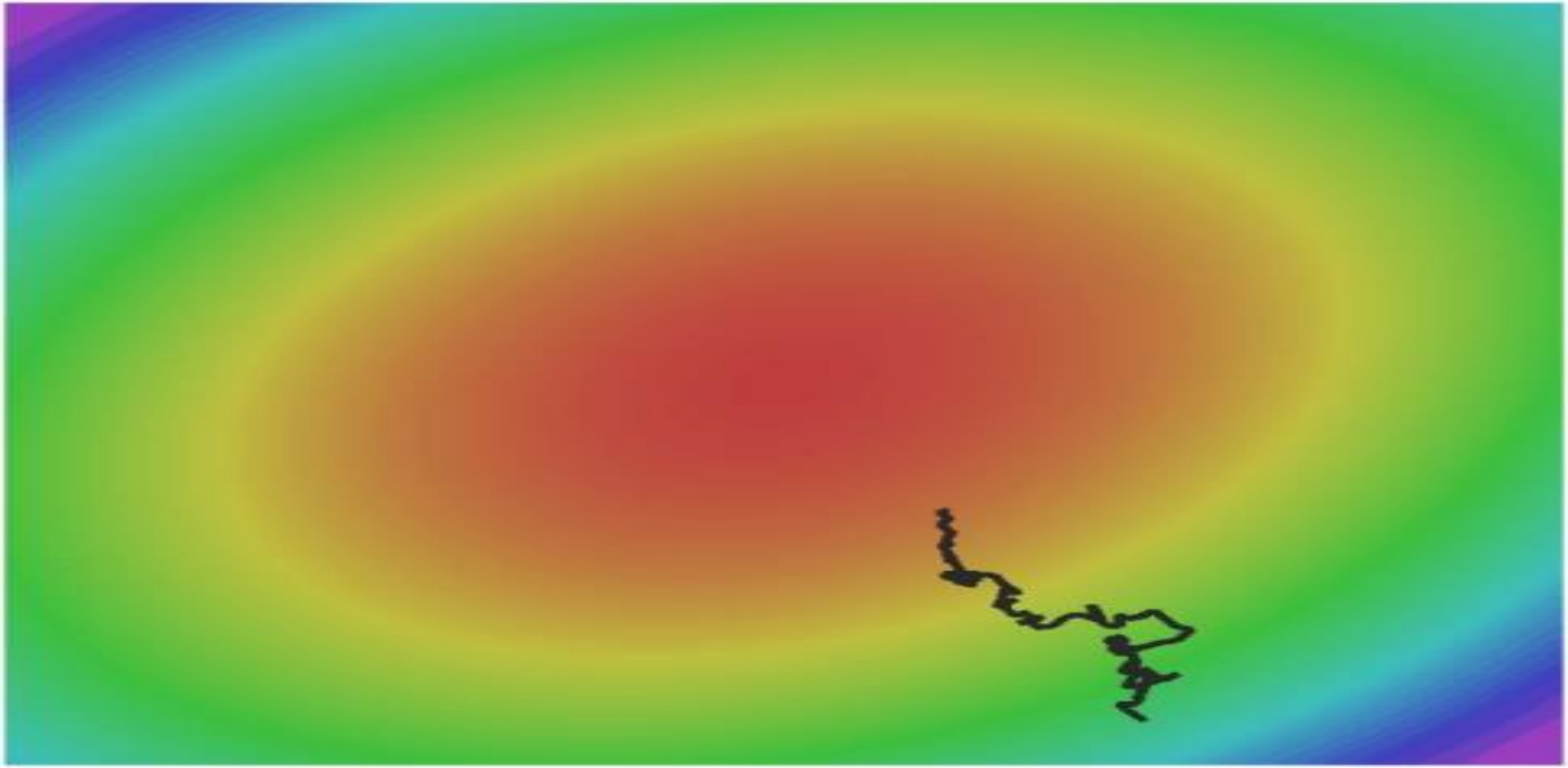
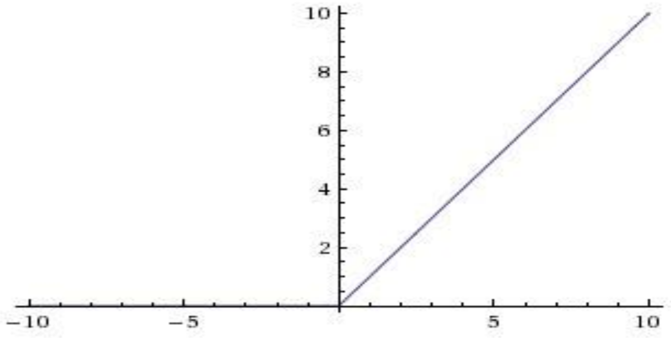


Image Source: cs231n, Stanford University

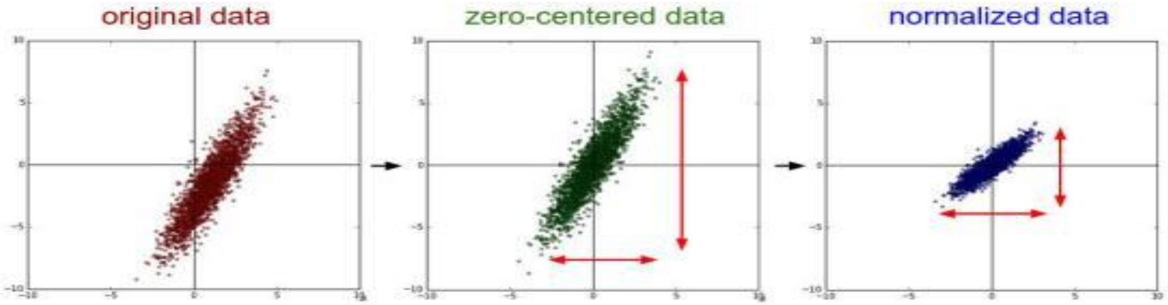
Previous Class

Training Aspects of CNN

- Activation Functions



- Dataset Preparation
- Data Preprocessing

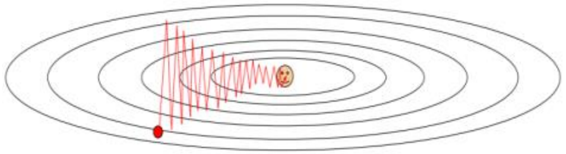


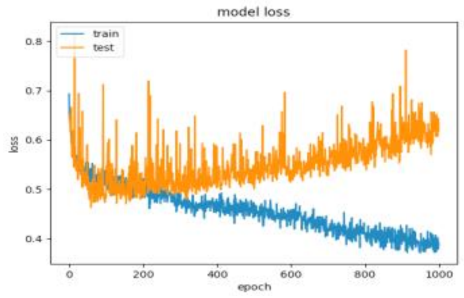
- Weight Initialization

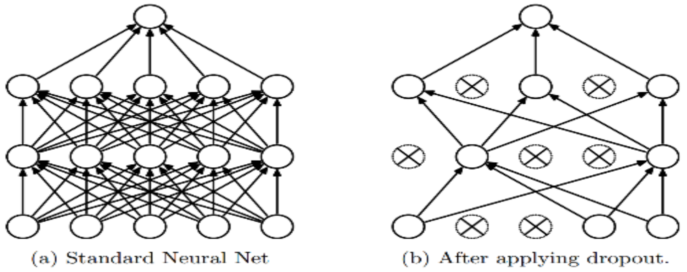
Next Few Classes

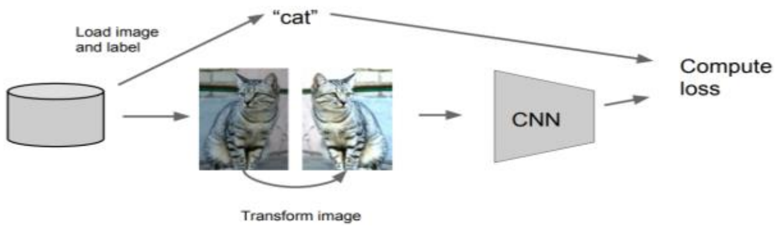
Training Aspects of CNN

- Optimization
- Learning Rate
- Regularization
- Dropout
- Batch Normalization
- Data Augmentation
- Transfer Learning
- Interpreting Loss Curve









Optimization

Source: cs231n

Mini-batch SGD

Loop:
1. **Sample** a batch of data
2. **Forward** prop it through the graph (network), get loss
3. **Backprop** to calculate the gradients
4. **Update** the parameters using the gradient

Source: cs231n

Stochastic Gradient Descent (SGD)

The procedure of repeatedly evaluating the gradient of loss function and then performing a parameter update.

Vanilla (Original) Gradient Descent:

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

Source: cs231n

SGD: Problems

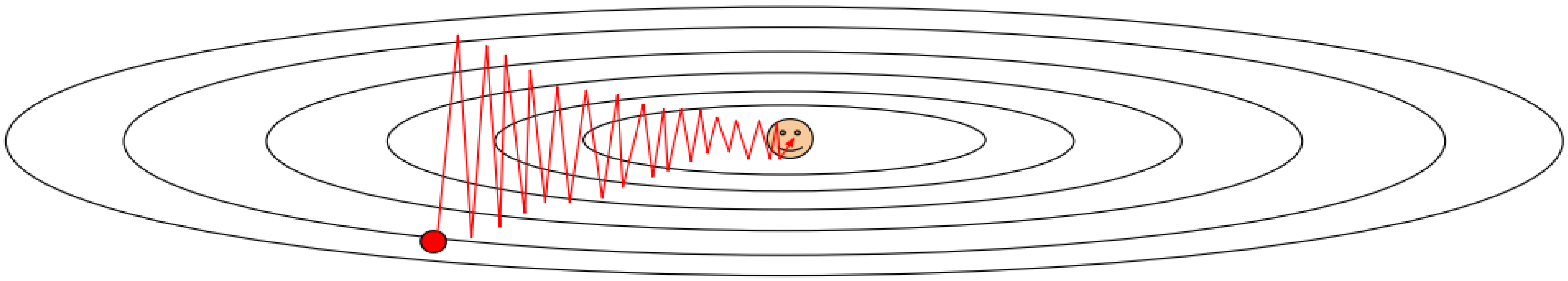
What if loss changes quickly in one direction and slowly in another?

Source: cs231n

SGD: Problems

What if loss changes quickly in one direction and slowly in another?

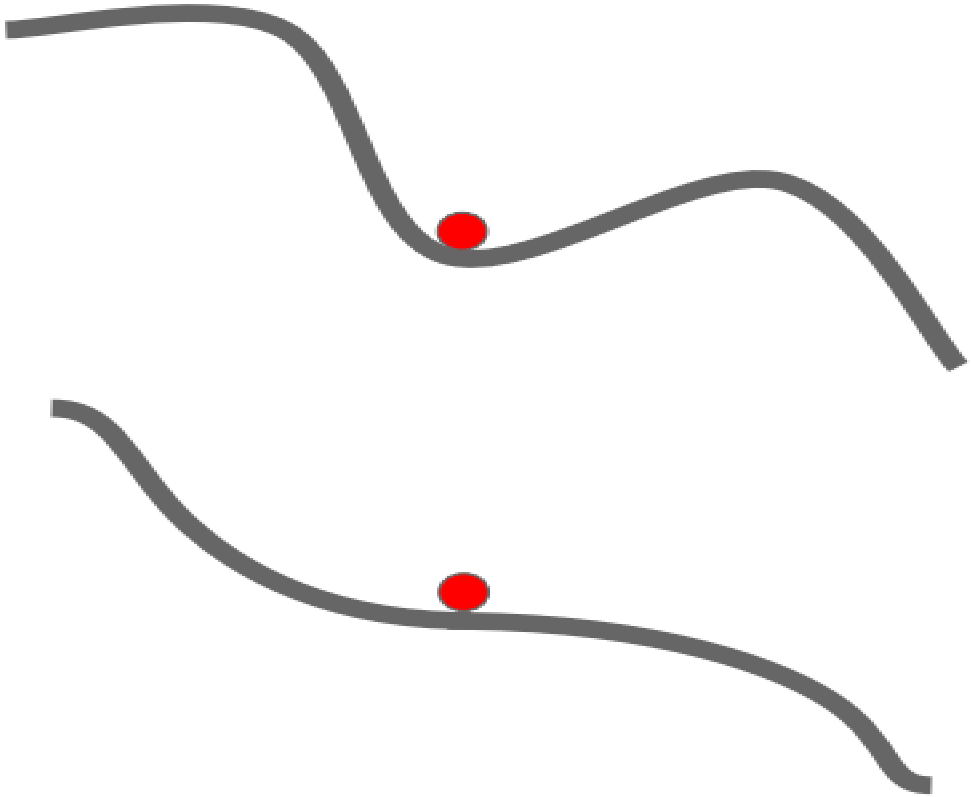
Very slow progress along shallow dimension, jitter along steep direction



Source: cs231n

SGD: Problems

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

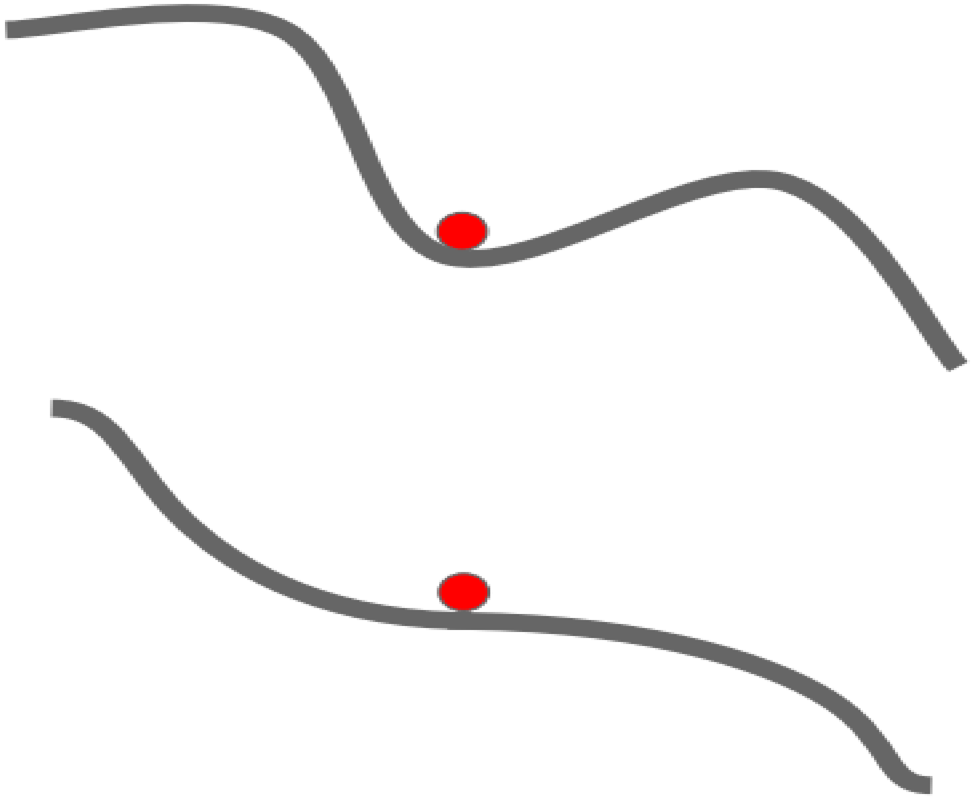


Source: cs231n

SGD: Problems

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

Zero gradient, gradient descent gets stuck



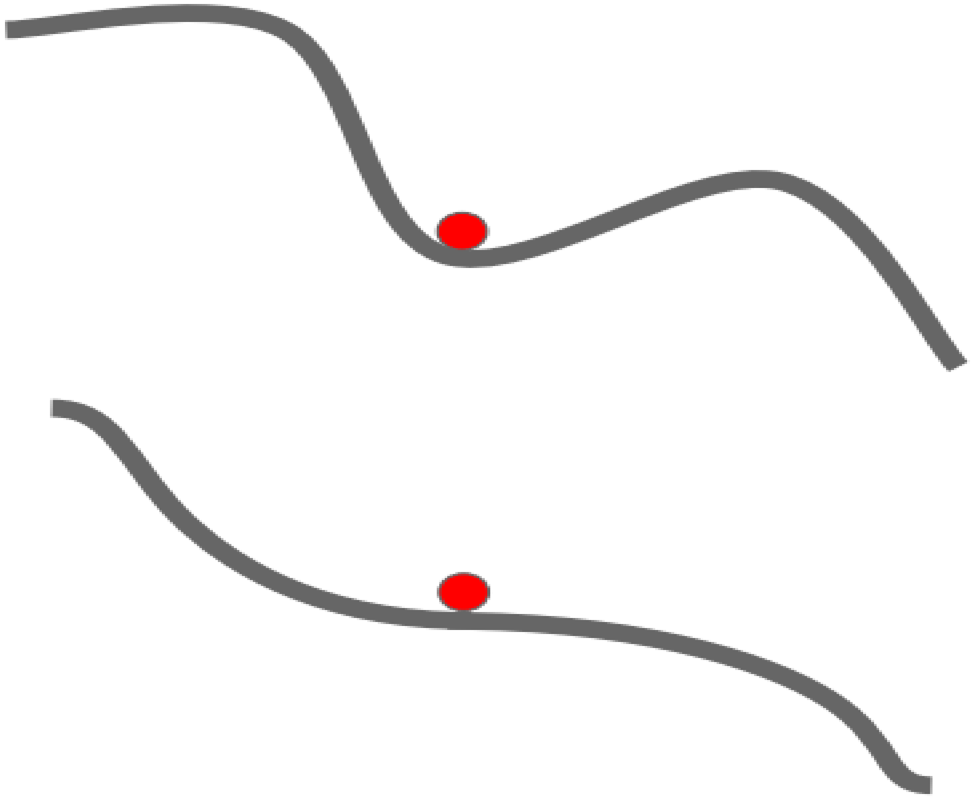
Source: cs231n

SGD: Problems

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

Zero gradient, gradient descent gets stuck

Saddle points much more common in high dimension



Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NIPS 2014

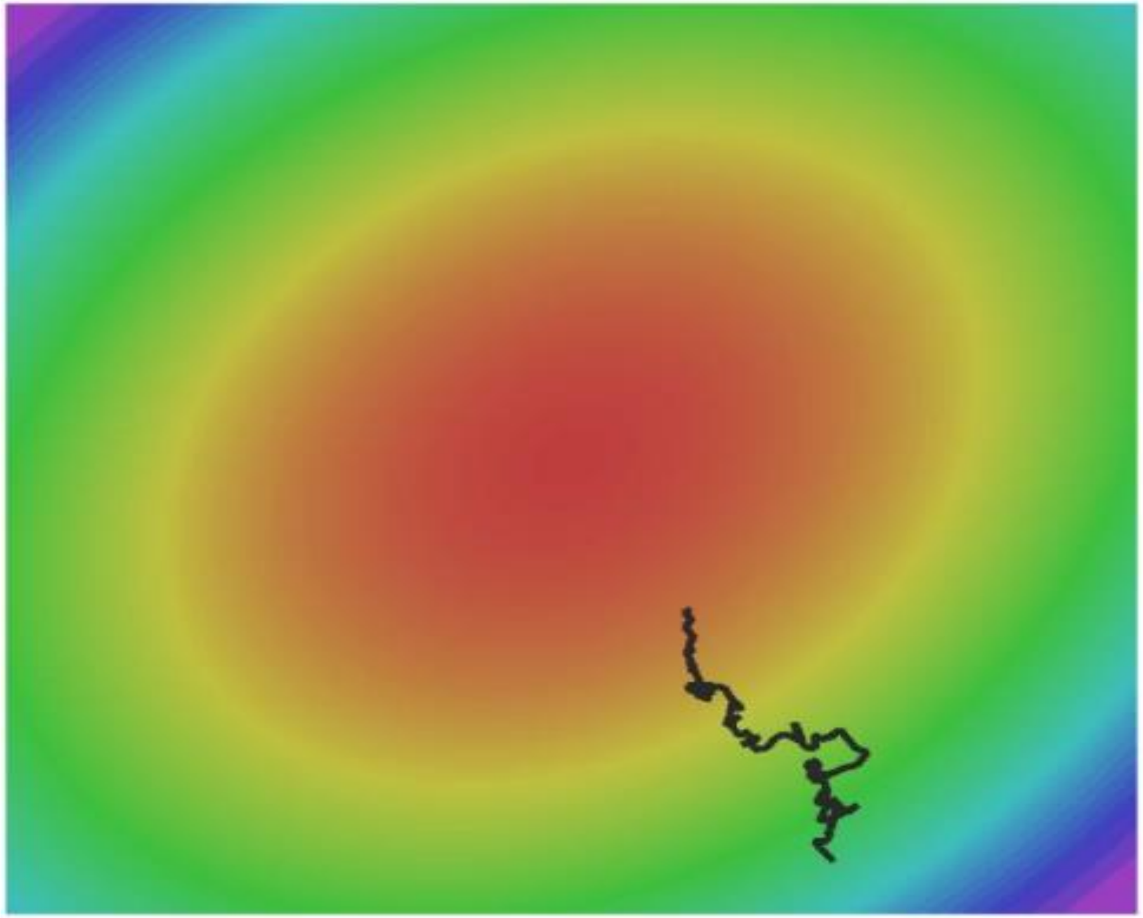
Source: cs231n

SGD: Problems

Our gradients come from **minibatches** so they can be **noisy!**

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

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



Source: cs231n

SGD + Momentum

SGD

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

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

Source: cs231n

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

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

Source: cs231n

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
```

Source: cs231n

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” in any direction that has consistent gradient
- Rho gives “friction”; typically rho=0.9 or 0.99

Source: cs231n

SGD + Momentum

SGD

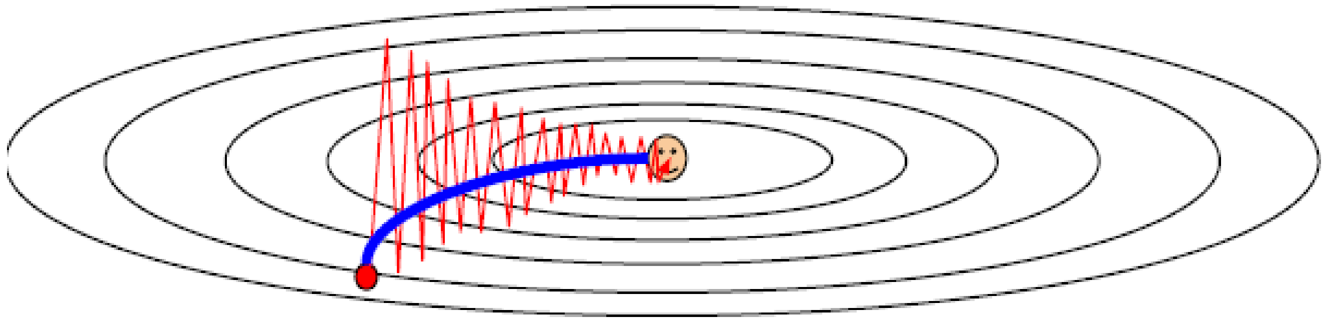
$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
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
```



Source: cs231n

AdaGrad

```
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

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

Source: cs231n

AdaGrad

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

What happens to the step size over long time?

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

Source: cs231n

AdaGrad

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

What happens to the step size over long time?

Effective learning rate diminishing problem

Duchi et al, “Adaptive subgradient methods for online learning and stochastic optimization”, JMLR 2011

Source: cs231n

RMSProp

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)
```

Tieleman and Hinton, 2012

Source: cs231n

Adam

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

```
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))
```

Source: cs231n

Adam

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

```
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))
```

Sort of like RMSProp with Momentum

Source: cs231n

Adam

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

```
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))
```

Sort of like RMSProp with Momentum

Problem:

Initially, second_moment=0 and beta2=0.999

After 1st iteration, second_moment -> close to zero

So, very large step for update of x

Source: cs231n

Adam (with Bias correction)

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

```
first_moment = 0
second_moment = 0
for t in range(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))
```

AdaGrad/ RMSProp

Bias Correction

Momentum

Bias correction for the fact that first and second moment estimates start at zero

Source: cs231n

Adam (with Bias correction)

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

```
first_moment = 0
second_moment = 0
for t in range(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))
```

AdaGrad/ RMSProp

Bias Correction

Momentum

Bias correction for the fact that first and second moment estimates start at zero

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

Source: cs231n

Major Problem with Adam

- Does not the optimization trajectory information such as short term gradient behavior

- Overshoots the optima

- Oscillates near the optima

diffGrad Optimizer

Solves the previously mentioned problems by incorporating the local gradient change as friction in effective learning rate.

High local gradient change → low friction → high learning rate

Small local gradient change → high friction → slow learning rate

S.R. Dubey et al. (2020), diffGrad: An Optimization Method for Convolutional Neural Networks, IEEE Transactions on Neural Network and Learning Systems.

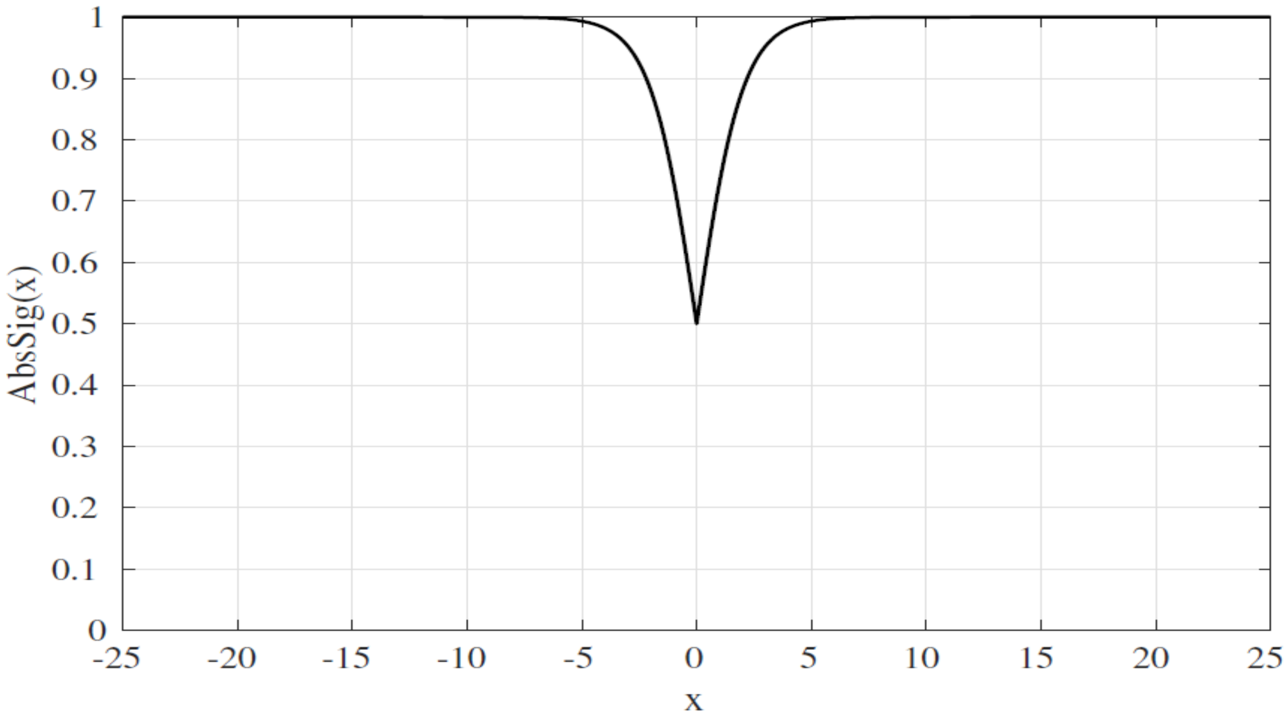
diffGrad Optimizer

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\alpha_t \times \xi_{t,i} \times \hat{m}_{t,i}}{\sqrt{\hat{v}_{t,i}} + \epsilon}$$

$$\xi_{t,i} = AbsSig(\Delta g_{t,i})$$

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

$$\Delta g_{t,i} = g_{t-1,i} - g_{t,i}$$



S.R. Dubey et al. (2020), diffGrad: An Optimization Method for Convolutional Neural Networks, IEEE Transactions on Neural Network and Learning Systems.

Recent SGD Based Optimizers

- Rectified Adam (RADAM)
- AdaBelief
- AngularGrad (Under Review) – by us
- AdaInject (Under Review) – by us

and many more.... still a challenging problem.

https://pythonawesome.com/a-collection-of-optimizers-for-pytorch/

Which optimizer to use in practice?

- Adaptive methods tend to reduce initial training error faster than SGD and are "safer"
  - Andrej Karpathy: *"In the early stages of setting baselines I like to use Adam with a learning rate of 3e-4. In my experience Adam is much more forgiving to hyperparameters, including a bad learning rate. For ConvNets a well-tuned SGD will almost always slightly outperform Adam, but the optimal learning rate region is much more narrow and problem-specific."*
  - Use Adam at first, then switch to SGD?

- However, some literature reports problems with adaptive methods, such as failing to converge or generalizing poorly (Wilson et al. 2017, Reddi et al. 2018)

Optimizer

In Practice:

- **Adam** is a good default choice in most cases
    - Try out RADAM, diffGrad and AdaBelief

More Optimizer: http://ruder.io/optimizing-gradient-descent/

Acknowledgement

Thanks to the following courses and corresponding researchers for making their teaching/research material online

- Deep Learning, Stanford University
- Introduction to Deep Learning, University of Illinois at Urbana-Champaign
- Introduction to Deep Learning, Carnegie Mellon University
- Convolutional Neural Networks for Visual Recognition, Stanford University
- Natural Language Processing with Deep Learning, Stanford University
- And Many More ......