
# CSE 176 Introduction to Machine Learning

## Lecture 11: Training Neural Network

Some slides from Fei-fei Li


# From last lectures:

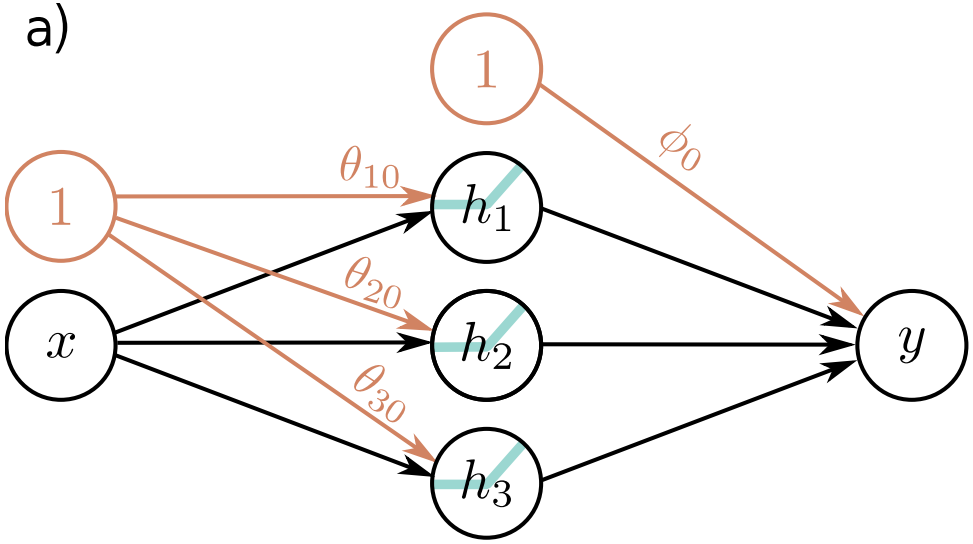
## Shallow & Deep Neural network, Losses, Optimization

# Depicting shallow neural networks

$$h_1 = \mathrm{a}[\theta_{10} + \theta_{11}x]$$
$$h_2 = \mathrm{a}[\theta_{20} + \theta_{21}x]$$
$$h_3 = \mathrm{a}[\theta_{30} + \theta_{31}x]$$

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

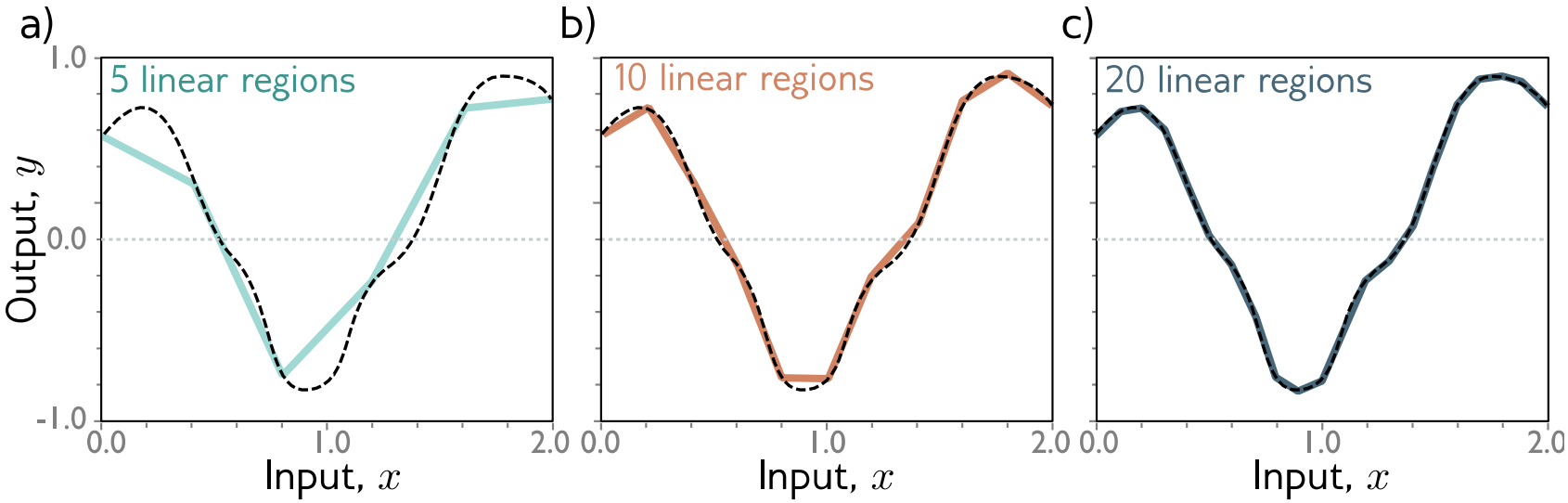
a)



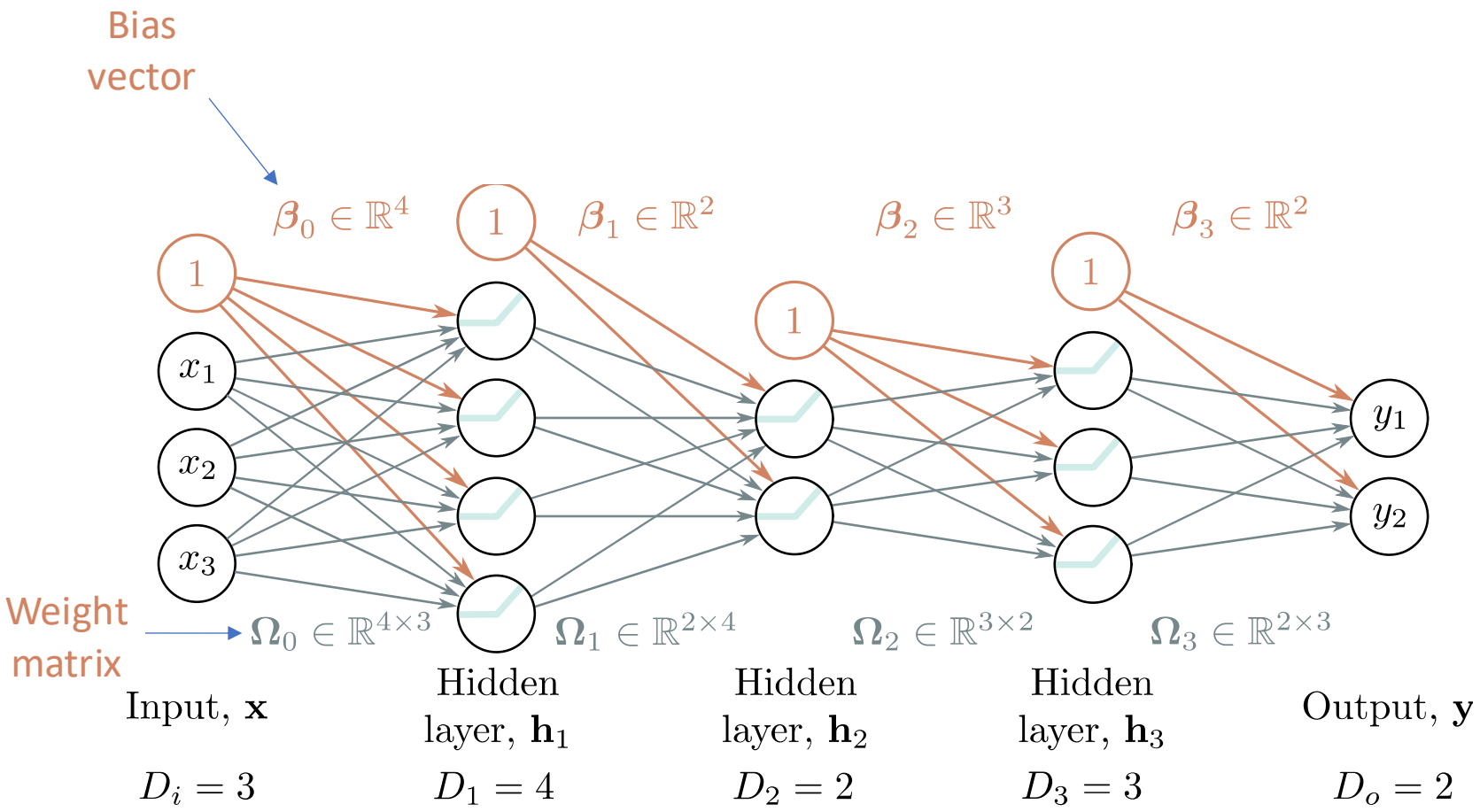

Each parameter multiplies its source and adds to its target

# With enough hidden units

❑… we can describe any 1D function to arbitrary accuracy



# Example of Multi Layer Perceptron (MLP)



# Cross-Entropy Loss (related to *logistic regression* loss)

Perceptron approximation: $\mathbf{f}(\mathbf{w}, \mathbf{x}^i) = u(W^T X^i) \approx \sigma(W^T X^i)$

Consider two probability distributions
over two classes (e.g. bass or salmon) : $(\mathbf{y}, 1-\mathbf{y})$ and $(\sigma, 1-\sigma)$


(binary)

**Cross-entropy loss:** $L(\mathbf{y}, \sigma) = -\mathbf{y} \ln \sigma - (1-\mathbf{y}) \ln(1-\sigma)$

(general multi-class case)

# Cross-Entropy Loss

K-label perceptron's output: $\bar{\sigma}(\mathbf{W}X^i)$ for example $X^i$ *k*-th index

Multi-valued label $\mathbf{y}^i = k$ gives **one-hot** distribution $\bar{\mathbf{y}}^i = (0, 0, 1, 0, \ldots, 0)$

Consider two probability distributions over K classes (e.g. bass, salmon, sturgeon): $\bar{\mathbf{y}}^i$ and $(\bar{\sigma}_1, \bar{\sigma}_2, \bar{\sigma}_3, \ldots, \bar{\sigma}_K)$


$$\Pr(\mathbf{x}^i \in \text{Class } k \mid W) = \bar{\sigma}_k(WX^i)$$

**cross entropy**

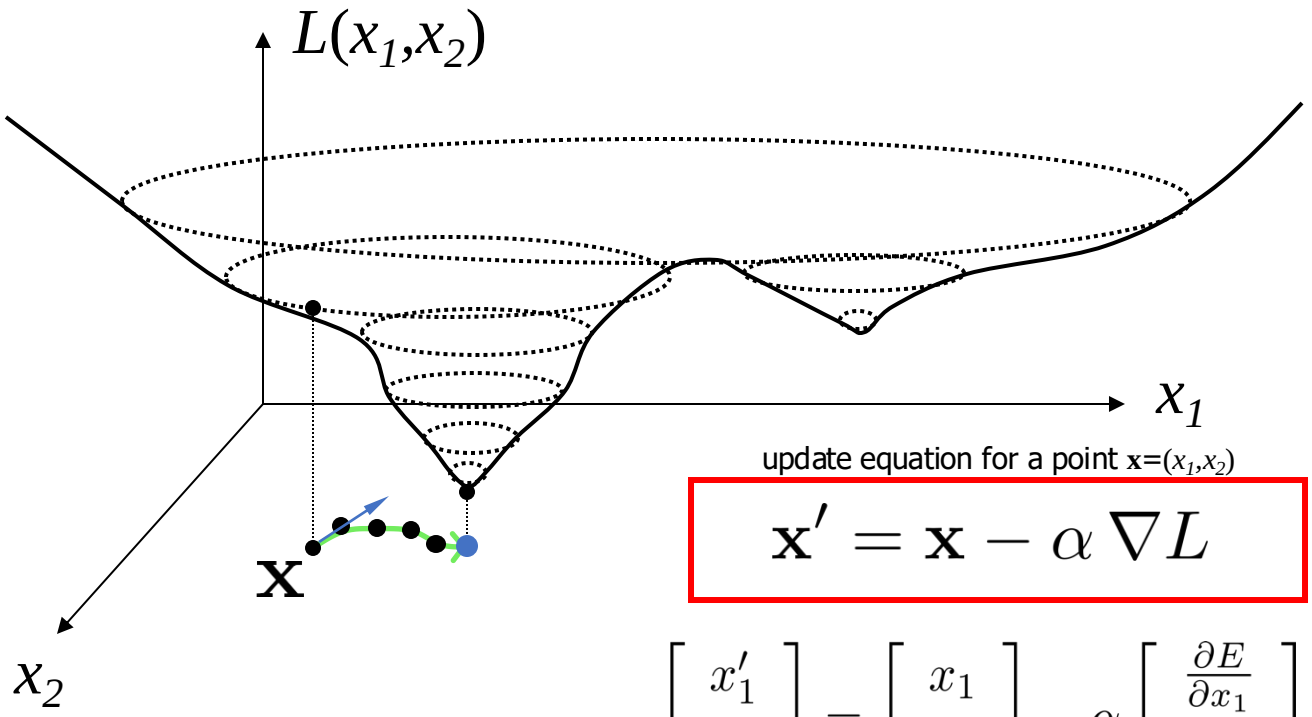
**Total loss:** $$L(W) = \sum_{i \in \text{train}} \sum_k -\bar{\mathbf{y}}^i_k \ln \bar{\sigma}_k(WX^i)$$

$$\Rightarrow \quad L(W) = -\sum_{i \in \text{train}} \ln \bar{\sigma}_{\mathbf{y}^i}(WX^i)$$

sum of **Negative Log-Likelihoods (NLL)**

# From last lecture: Gradient Descent

- Example: for a function of two variables



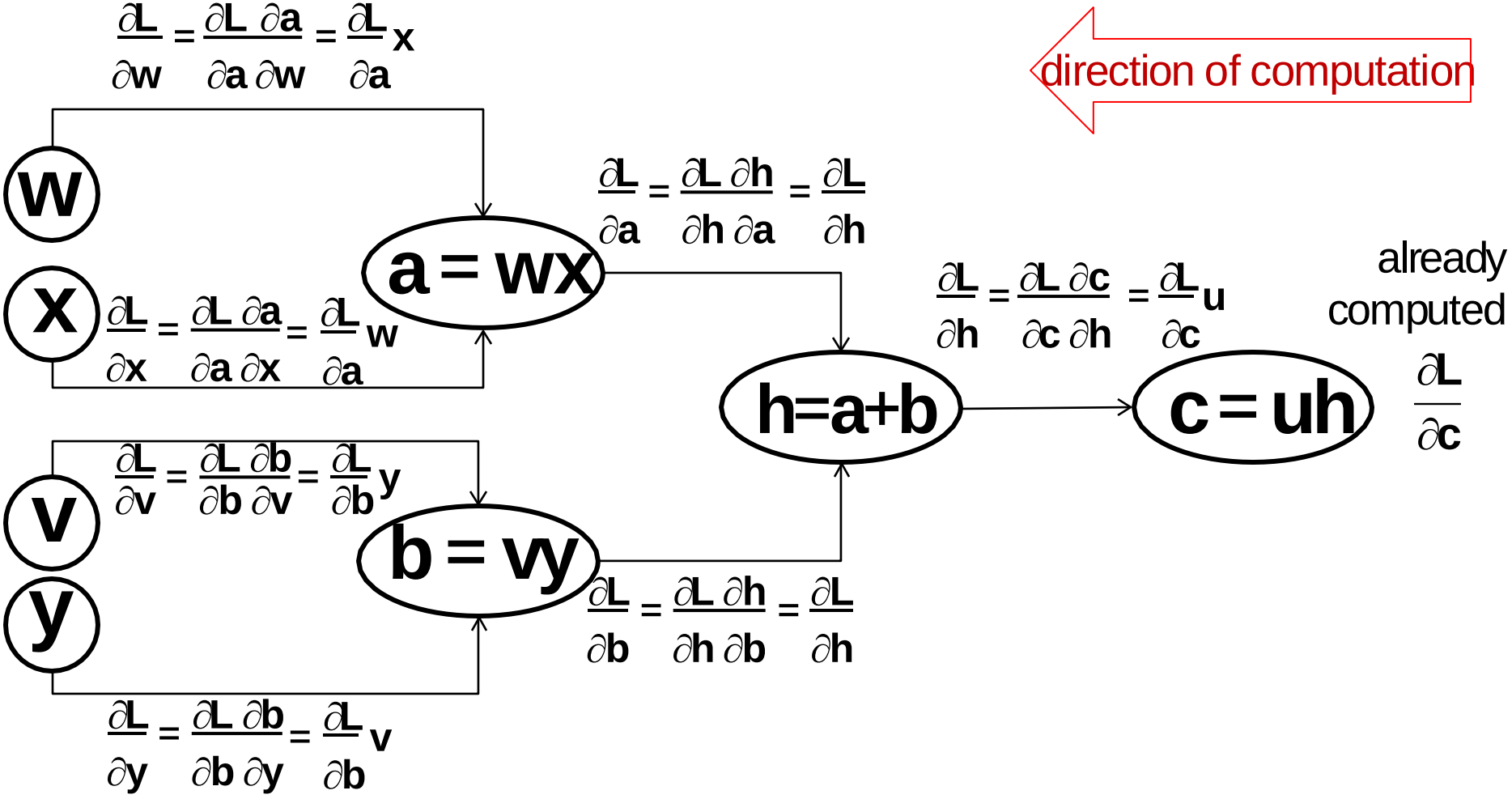

update equation for a point $\mathbf{x}=(x_1,x_2)$

$$\mathbf{x}' = \mathbf{x} - \alpha \nabla L$$

$$\begin{bmatrix} x_1' \\ x_2' \end{bmatrix} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} - \alpha \begin{bmatrix} \frac{\partial E}{\partial x_1} \\ \frac{\partial E}{\partial x_2} \end{bmatrix}$$

Stop at a local minima where $\nabla L = \vec{0}$

# From last lecture: Backpropogation




- Some of these partial derivatives are intermediate
  - their values will not be used for gradient descent

# Today

- ❑ Deep learning hardware
- ❑ Deep learning software
- ❑ Tricks for training neural networks
  - ❑ Activation function
  - ❑ Data Preprocessing
  - ❑ Batch normalization
  - ❑ Transfer learning

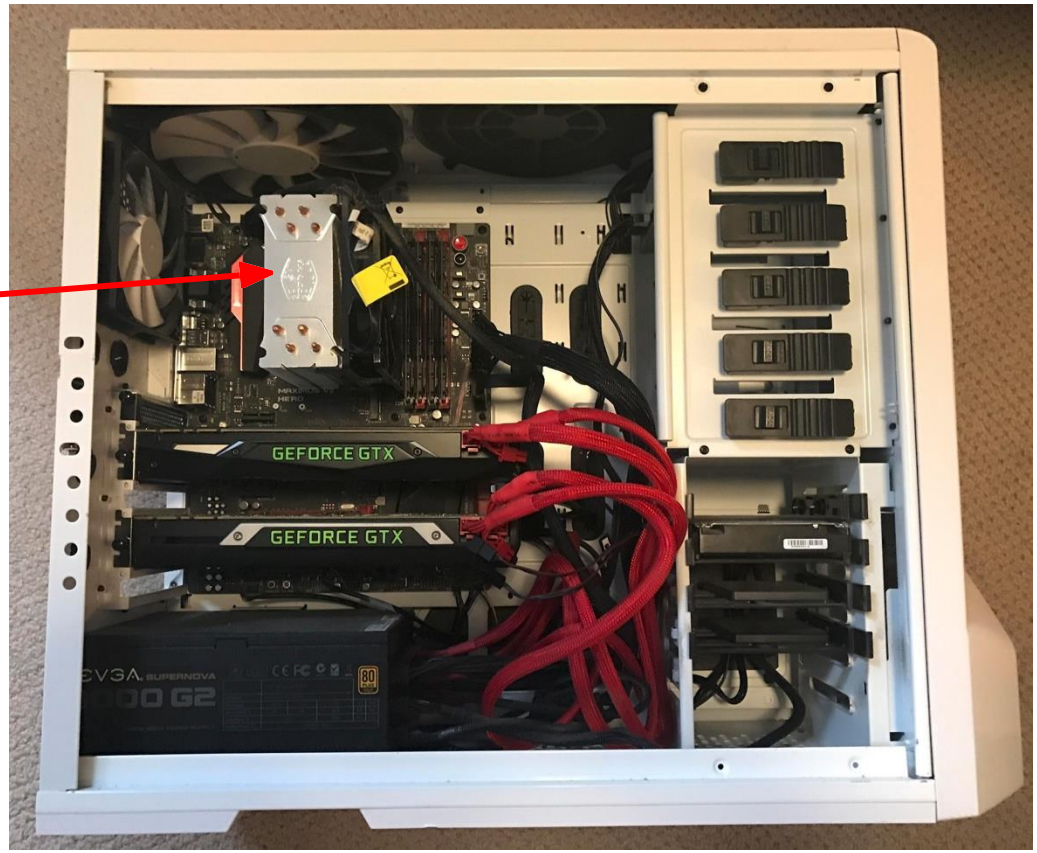

# Deep Learning Hardware

# Inside a computer

# Spot the CPU!

(central processing unit)





# Spot the GPUs!

(graphics processing unit)



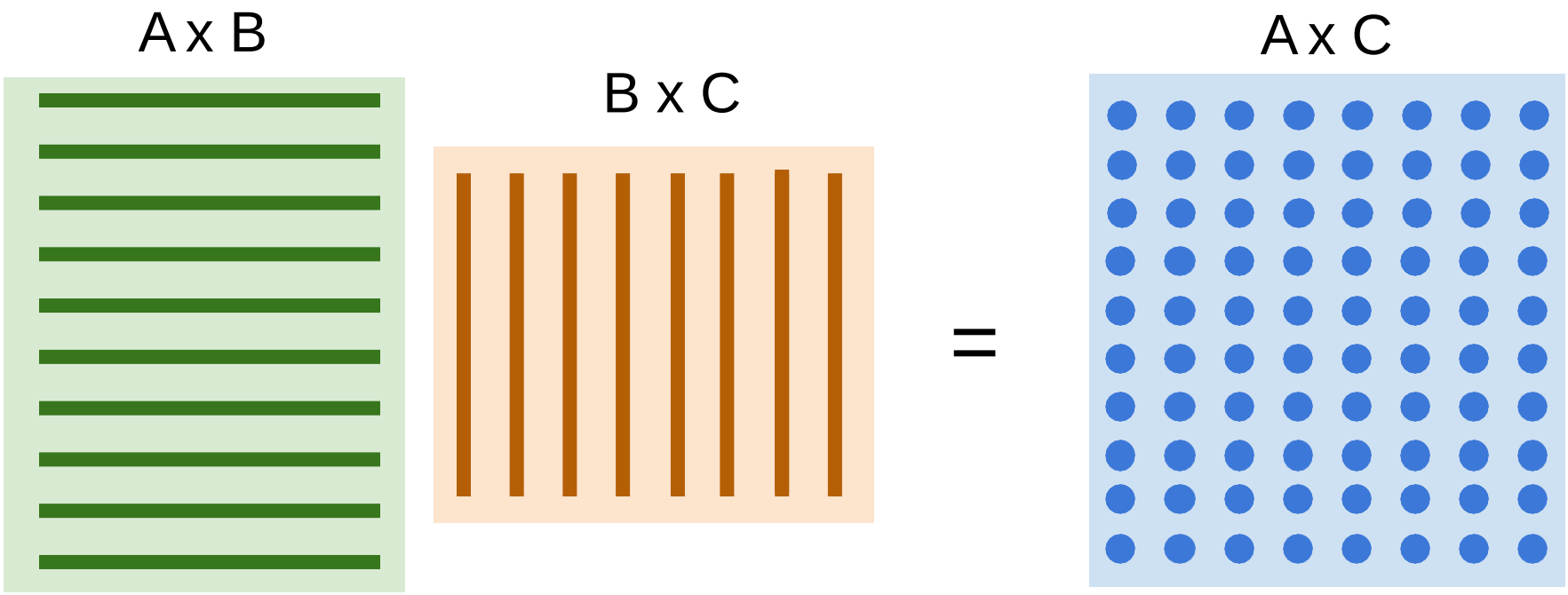

# CPU vs GPU

| | Cores | Clock Speed | Memory | Price | Speed (throughput) |
|---|---|---|---|---|---|
| **CPU** (Intel Core i9-7900k) | 10 | 4.3 GHz | System RAM | $385 | ~640 **G**FLOPS FP32 |
| **GPU** (NVIDIA RTX 3090) | 10496 | 1.6 GHz | 24 GB GDDR6X | $1499 | ~35.6 **T**FLOPS FP32 |

**CPU**: Fewer cores, but each core is much faster and much more capable; great at sequential tasks

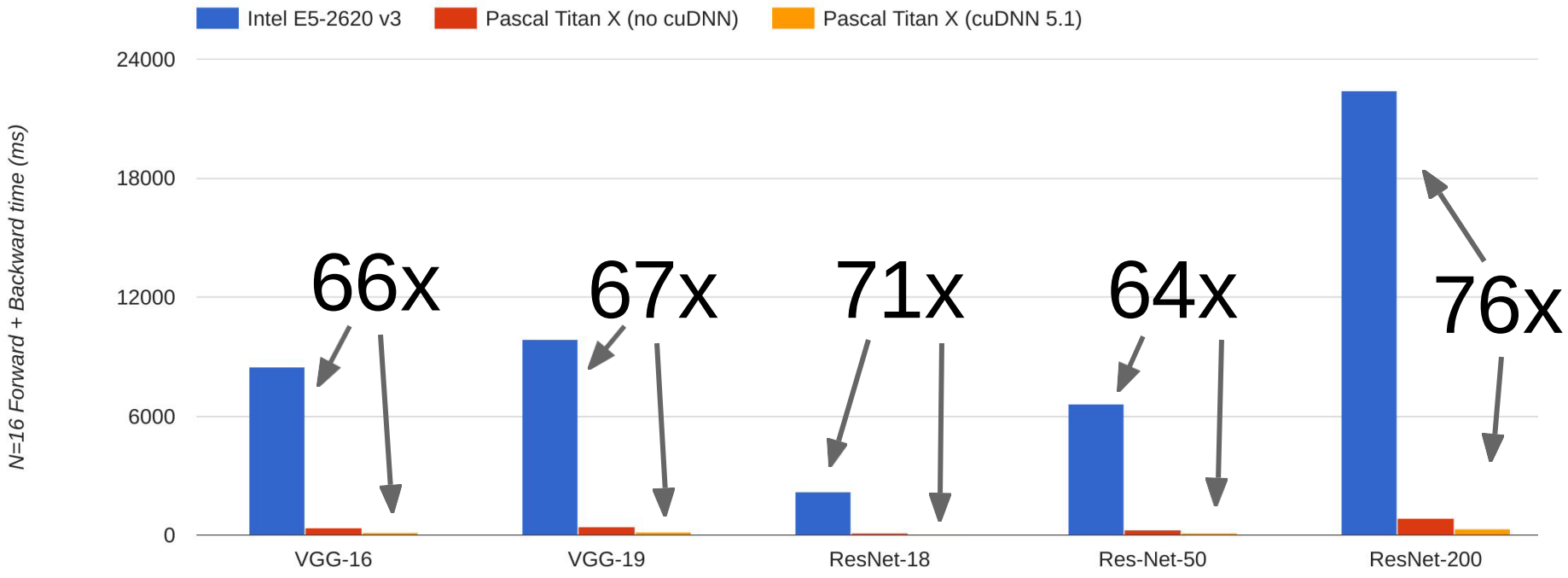
**GPU**: More cores, but each core is much slower and “dumber”; great for parallel tasks

# Example: Matrix Multiplication

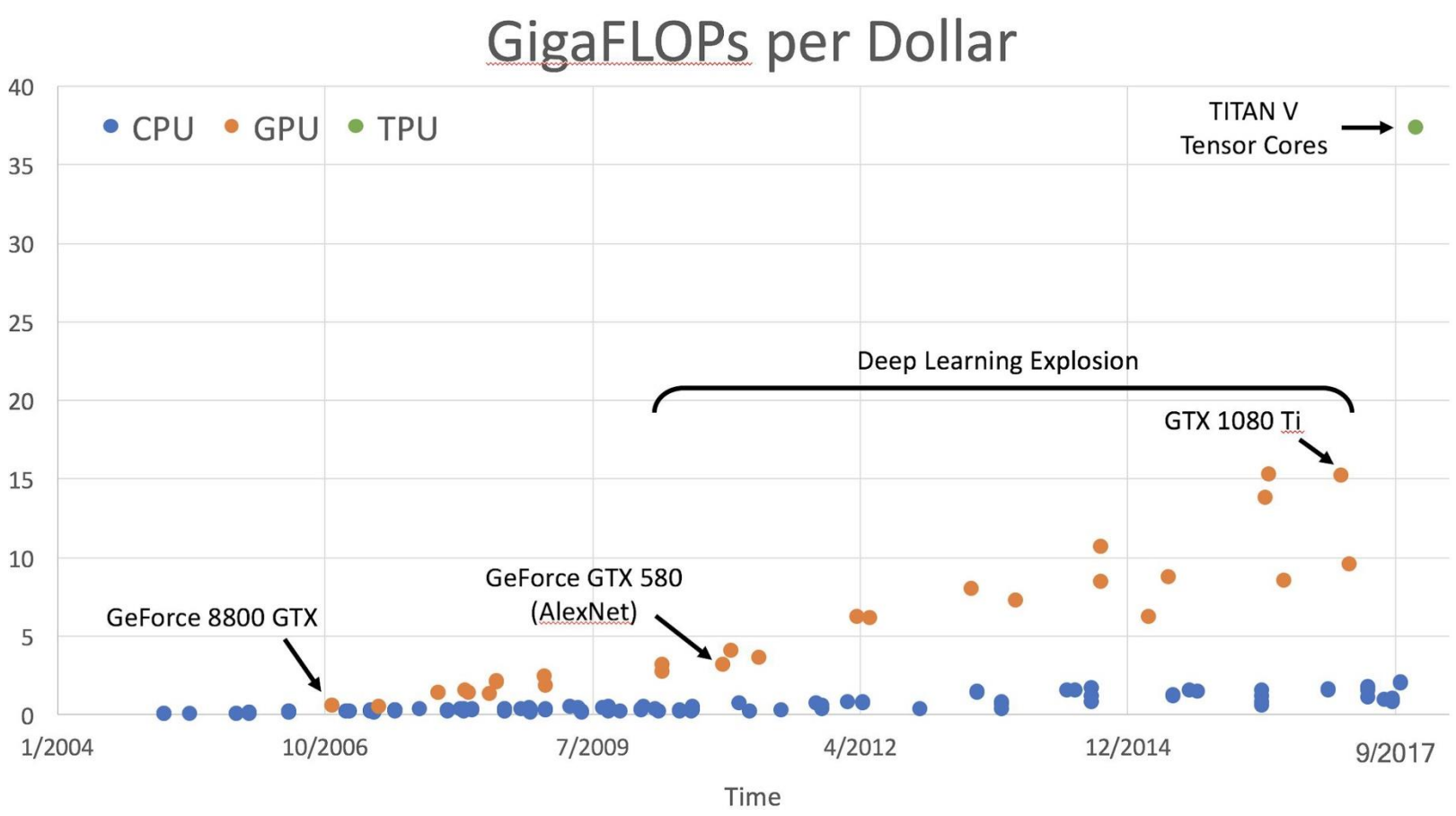



cuBLAS::GEMM (GEneral Matrix-to-matrix Multiply)

# CPU vs GPU in practice

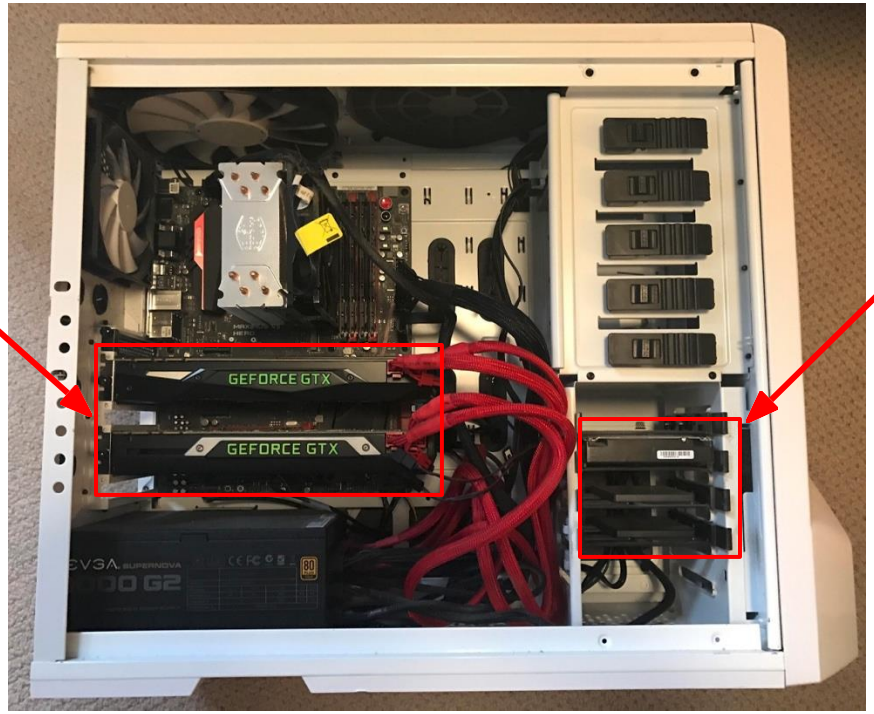



Data from https://github.com/jcjohnson/cnn-benchmarks


GigaFLOPs per Dollar
CPU
GPU
TPU
TITAN V
Tensor Cores
Deep Learning Explosion
GTX 1080 Ti
GeForce GTX 580
(AlexNet)
GeForce 8800 GTX
40
35
30
25
20
15
10
5
0
1/2004
10/2006
7/2009
4/2012
12/2014
9/2017
Time

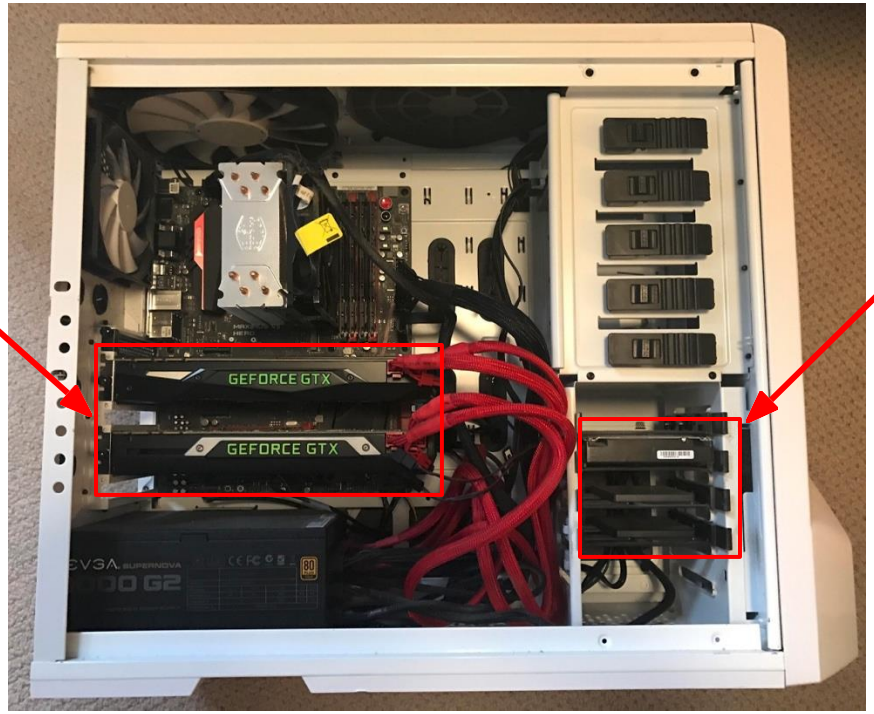
# CPU / GPU Communication

Model is here




Data is here

# CPU / GPU Communication




If you aren't careful, training can bottleneck on reading data and transferring to GPU!

**Solutions**:

- Read all data into RAM
- Use SSD instead of HDD
- Use multiple CPU threads to prefetch data


# Deep Learning Software

# A zoo of frameworks!

Caffe
(UC Berkeley)

Caffe2
(Facebook)
mostly features absorbed by PyTorch

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

TensorFlow
(Google)

PaddlePaddle
(Baidu)

Chainer
(Preferred Networks)
The company has officially migrated its research infrastructure to PyTorch

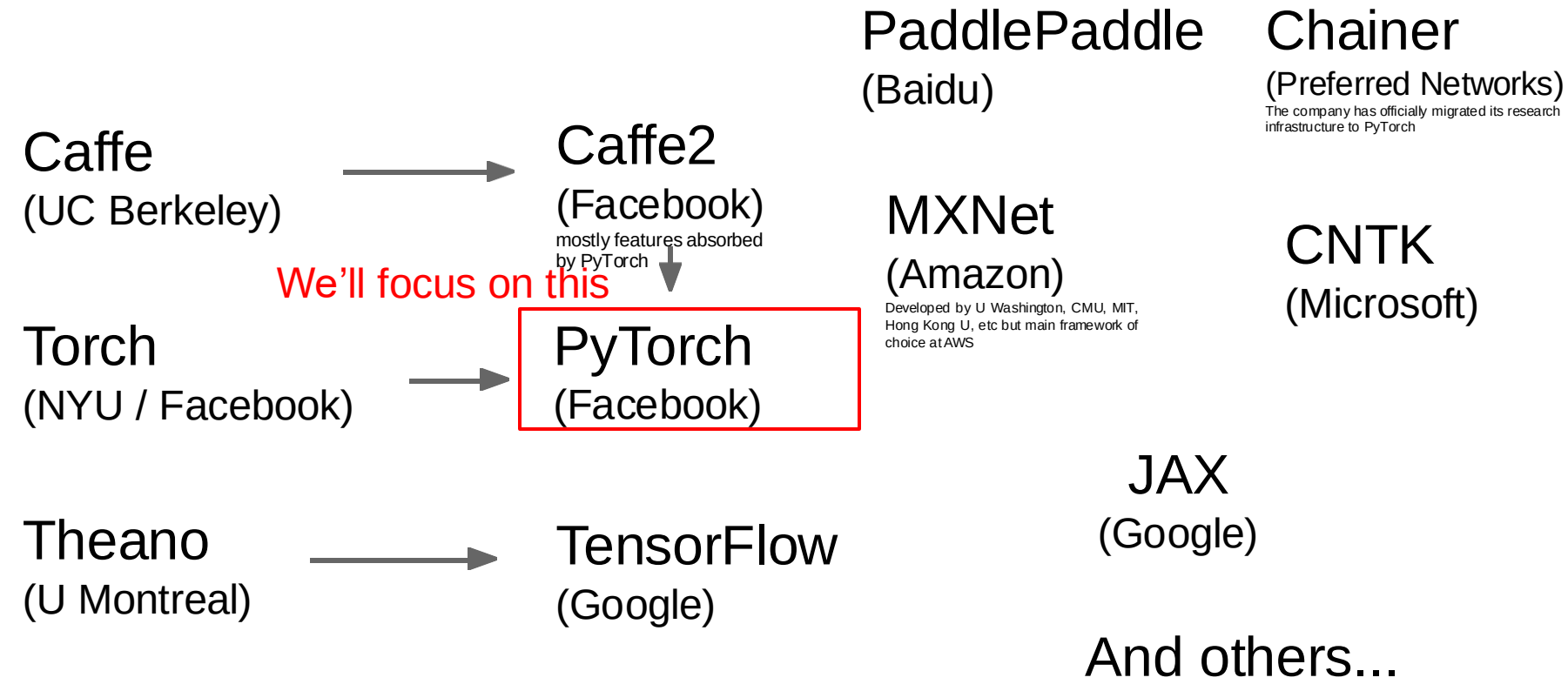
MXNet
(Amazon)
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

JAX
(Google)

And others...

# A zoo of frameworks!



# Deep Learning Framework

(1) Quick to develop and test new ideas

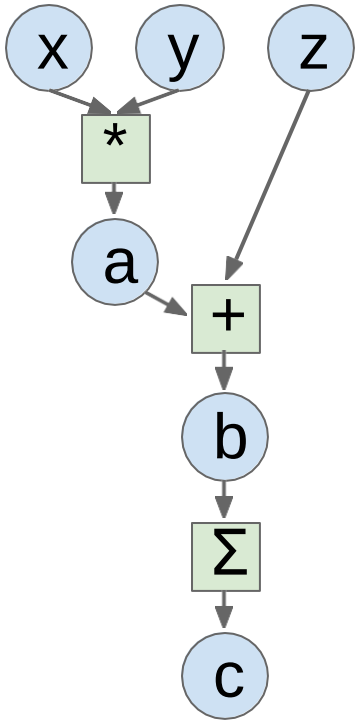
(2) Automatically compute gradients

(3) Run it all efficiently on GPU (wrap cuDNN, cuBLAS, OpenCL, etc)

# Computational Graphs

## Numpy

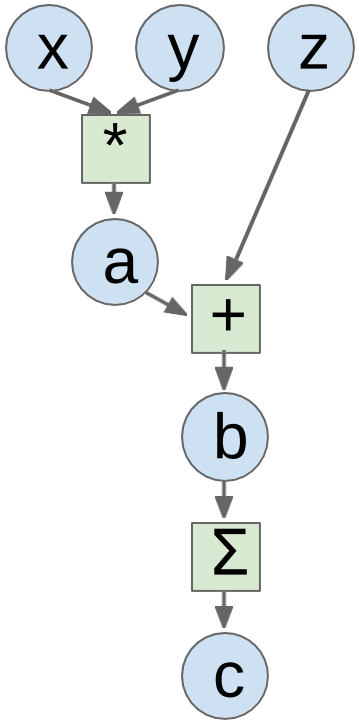
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```



# Computational Graphs

## Numpy

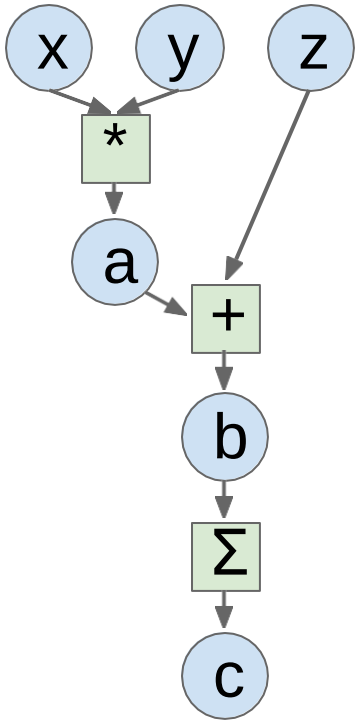
```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```



# Computational Graphs

## Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```




**Good**:

Clean API, easy to write numeric code

**Bad**:

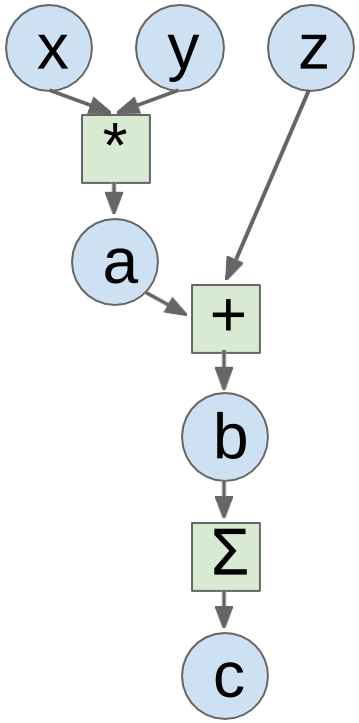
- Have to compute our own gradients
- Can't run on GPU

# Computational Graphs

## Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)
```

```
grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```




## PyTorch

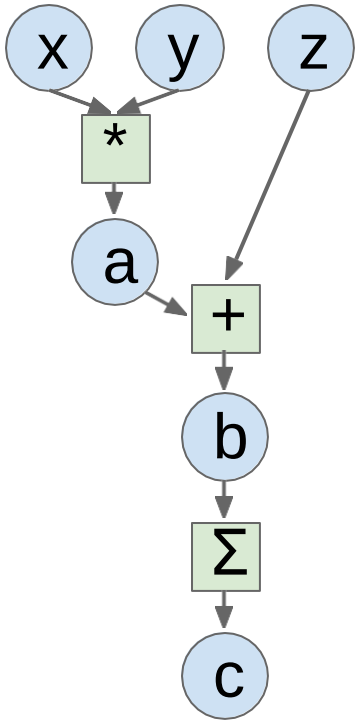
```
import torch

N, D = 3, 4
x = torch.randn(N, D)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)
```

Looks exactly like numpy!

# Computational Graphs

## Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```




## PyTorch

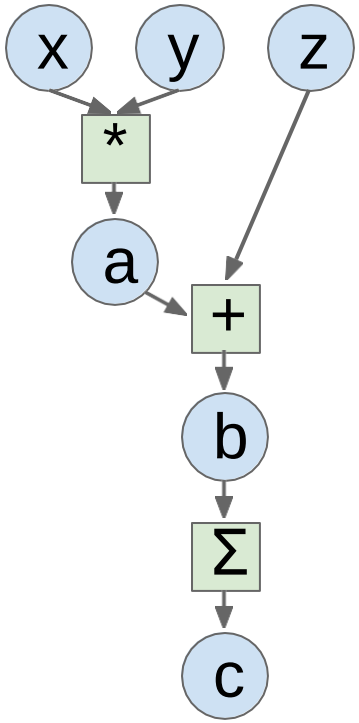
```
import torch

N, D = 3, 4
x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D)
z = torch.randn(N, D)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

PyTorch handles gradients for us!

# Computational Graphs

## Numpy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```




## PyTorch

```
import torch

device = 'cuda:0'
N, D = 3, 4
x = torch.randn(N, D, requires_grad=True,
                device=device)
y = torch.randn(N, D, device=device)
z = torch.randn(N, D, device=device)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
print(x.grad)
```

Trivial to run on GPU - just construct arrays on a different device!

# PyTorch
(More details)

# Pytorch fundamental concepts

- **torch.Tensor**: Like a numpy array, but can run on GPU

- **torch.autograd**: Package for building computational graphs out of Tensors, and automatically computing gradients

- **torch.nn.Module**: A neural network layer; may store state or learnable weights

# Pytorch:Tensor

Running example: Train a two-layer ReLU network on random data with L2 loss

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

UCMERCED

# Pytorch:Tensor

Create random tensors
for data and weights



```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# Pytorch:Tensor

Forward pass: compute predictions and loss



```python
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# Pytorch:Tensor

Backward pass: manually compute gradients



```python
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# Pytorch:Tensor

```
import torch

device = torch.device('cpu')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

Gradient descent step on weights

# Pytorch:Tensor

To run on GPU, just use a different device!

```
import torch

device = torch.device('cuda:0')

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in, device=device)
y = torch.randn(N, D_out, device=device)
w1 = torch.randn(D_in, H, device=device)
w2 = torch.randn(H, D_out, device=device)

learning_rate = 1e-6
for t in range(500):
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)
    loss = (y_pred - y).pow(2).sum()

    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2
```

# Pytorch:Autograd

Creating Tensors with requires_grad=True enables autograd

Operations on Tensors with requires_grad=True cause PyTorch to build a computational graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

# Pytorch:Autograd

Forward pass looks exactly the same as before, but we don't need to track intermediate values - PyTorch keeps track of them for us in the graph

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

# Pytorch:Autograd

Compute gradient of loss with respect to w1 and w2

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

# Pytorch:Autograd

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

Make gradient step on weights, then zero them. Torch.no_grad means “don’t build a computational graph for this part”

# Pytorch:Autograd

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)
w1 = torch.randn(D_in, H, requires_grad=True)
w2 = torch.randn(H, D_out, requires_grad=True)

learning_rate = 1e-6
for t in range(500):
    y_pred = x.mm(w1).clamp(min=0).mm(w2)
    loss = (y_pred - y).pow(2).sum()

    loss.backward()

    with torch.no_grad():
        w1 -= learning_rate * w1.grad
        w2 -= learning_rate * w2.grad
        w1.grad.zero_()
        w2.grad.zero_()
```

PyTorch methods that end in underscore modify the Tensor in-place; methods that don't return a new Tensor

# PyTorch: nn

Higher-level wrapper for working with neural nets

Use this! It will make your life easier

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

# PyTorch: nn

Define our model as a sequence of layers; each layer is an object that holds learnable weights



```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

# PyTorch: nn

Forward pass: feed data to model, and compute loss

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

# PyTorch: nn

Forward pass: feed data to model, and compute loss

torch.nn.functional has useful helpers like loss functions

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

# PyTorch: nn

Backward pass: compute gradient with respect to all model weights (they have requires_grad=True)



```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

# PyTorch: nn

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-2
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * param.grad
    model.zero_grad()
```

Make gradient step on each model parameter (with gradients disabled)

# PyTorch: nn

Use an **optimizer** for different update rules

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

# PyTorch: nn

```
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = torch.nn.Sequential(
          torch.nn.Linear(D_in, H),
          torch.nn.ReLU(),
          torch.nn.Linear(H, D_out))

learning_rate = 1e-4
optimizer = torch.optim.Adam(model.parameters(),
                             lr=learning_rate)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()

    optimizer.step()
    optimizer.zero_grad()
```

After computing gradients, use optimizer to update params and zero gradients

# PyTorch: nn
# Define new Modules

A PyTorch **Module** is a neural net layer; it inputs and outputs Tensors

Modules can contain weights or other modules

You can define your own Modules using autograd!

```
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

# PyTorch: nn
# Define new Modules

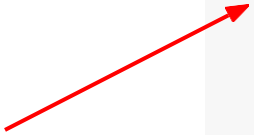
Define our whole model
as a single Module



```python
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

# PyTorch: nn
# Define new Modules

Initializer sets up two children (Modules can contain modules)



```python
import torch

class TwoLayerNet(torch.nn.Module):
    def __init__(self, D_in, H, D_out):
        super(TwoLayerNet, self).__init__()
        self.linear1 = torch.nn.Linear(D_in, H)
        self.linear2 = torch.nn.Linear(H, D_out)

    def forward(self, x):
        h_relu = self.linear1(x).clamp(min=0)
        y_pred = self.linear2(h_relu)
        return y_pred

N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

model = TwoLayerNet(D_in, H, D_out)

optimizer = torch.optim.SGD(model.parameters(), lr=1e-4)
for t in range(500):
    y_pred = model(x)
    loss = torch.nn.functional.mse_loss(y_pred, y)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

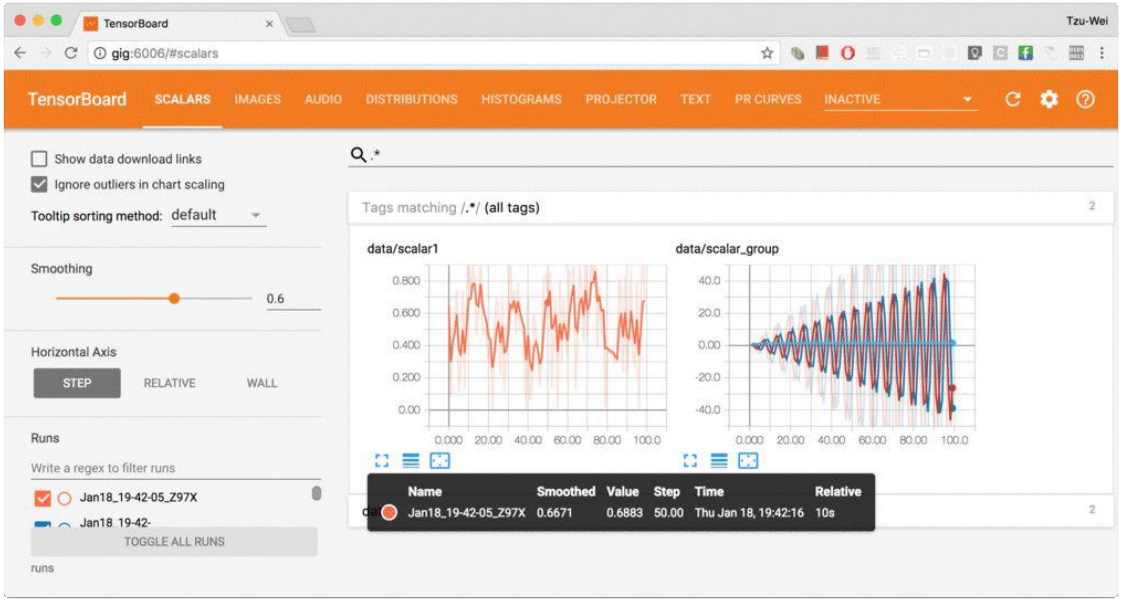
# PyTorch: Pretrained Models

Super easy to use pretrained models with torchvision
https://github.com/pytorch/vision

```
import torch
import torchvision

alexnet = torchvision.models.alexnet(pretrained=True)
vgg16 = torchvision.models.vgg16(pretrained=True)
resnet101 = torchvision.models.resnet101(pretrained=True)
```

# PyTorch: torch.utils.tensorboard

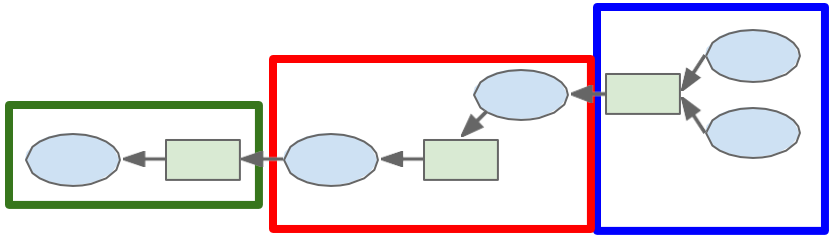
A python wrapper around Tensorflow's web-based visualization tool.



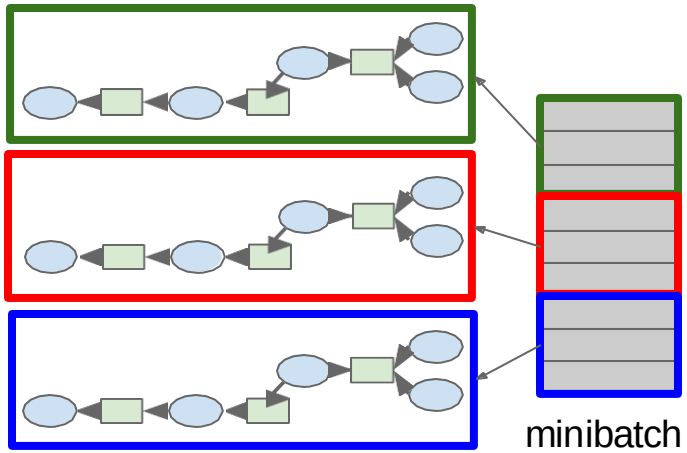
# Model Parallel vs. Data Parallel

Model parallelism: split computation graph into parts & distribute to GPUs/ nodes

Data parallelism: split minibatch into chunks & distribute to GPUs/ nodes



Model Parallel

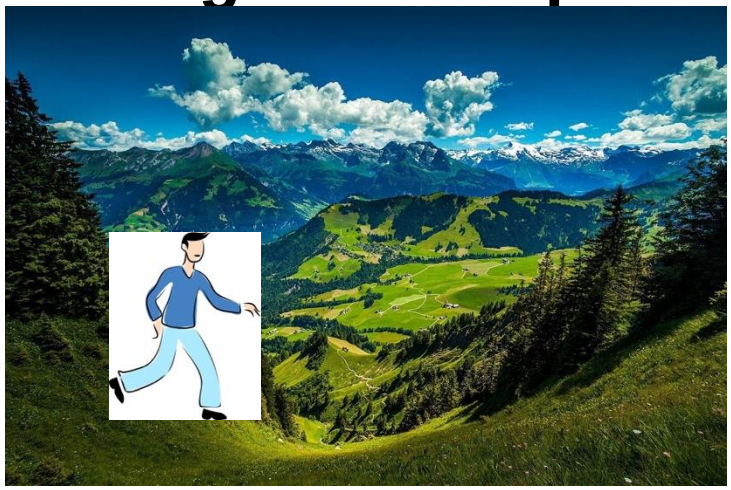


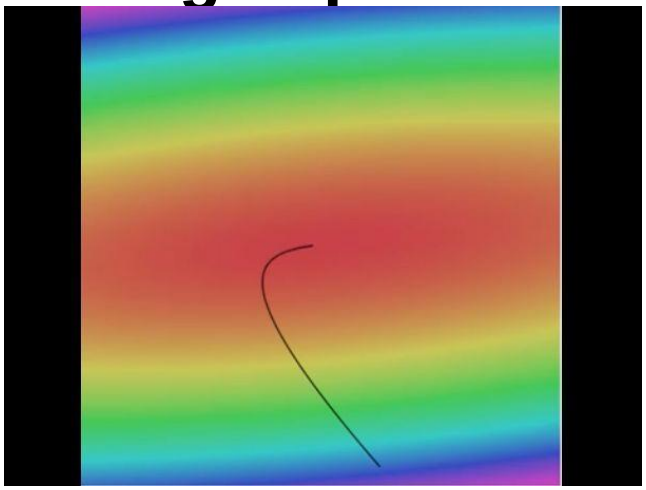

Data Parallel


# Tricks for training neural networks

# Where we are now...

## Learning network parameters through optimization





```
# Vanilla Gradient Descent

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

Where we are now...

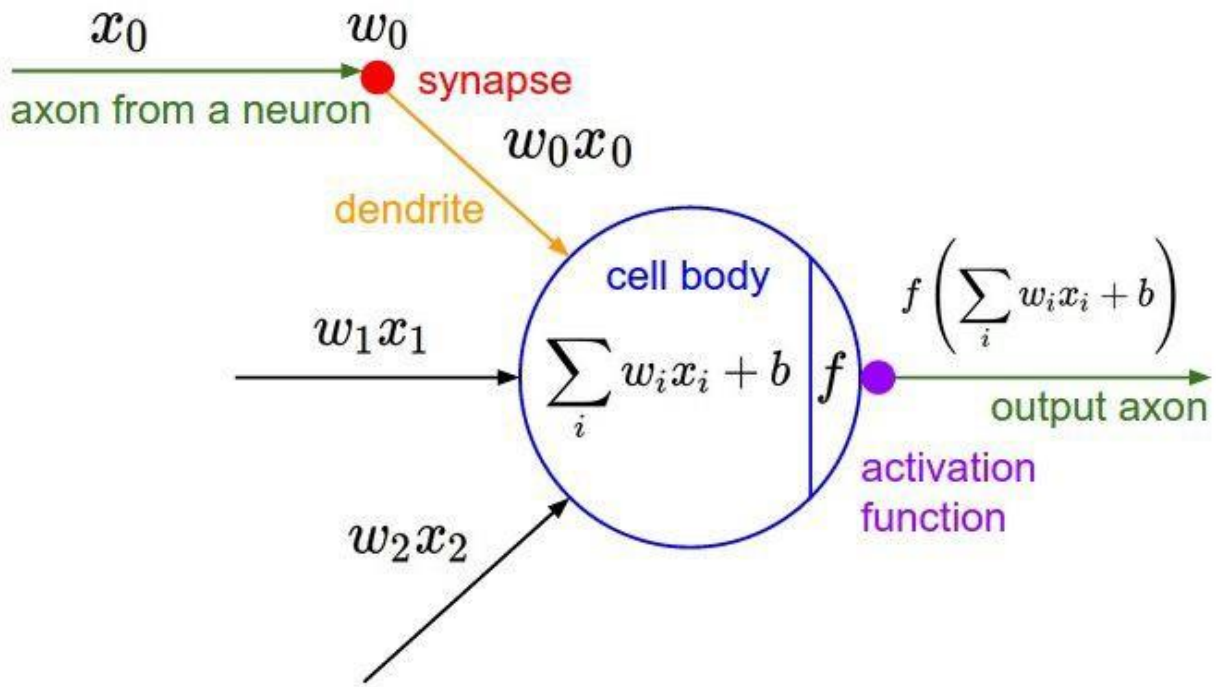
# Mini-batch SGD

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

# Activation Functions

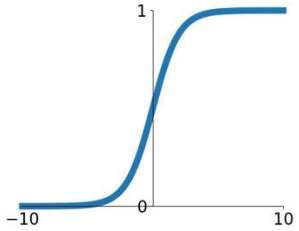
# Activation Functions



# Activation Functions

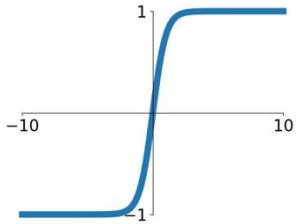
**Sigmoid**

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



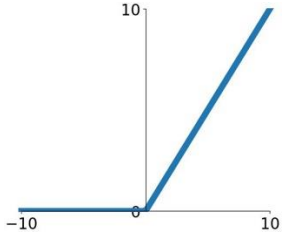

**tanh**

$$\tanh(x)$$



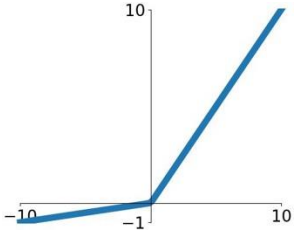

**ReLU**

$$\max(0, x)$$




**Leaky ReLU**

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

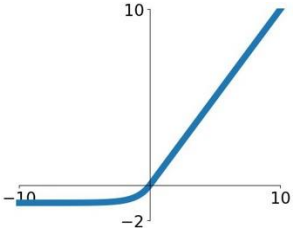



**Maxout**

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

**ELU**

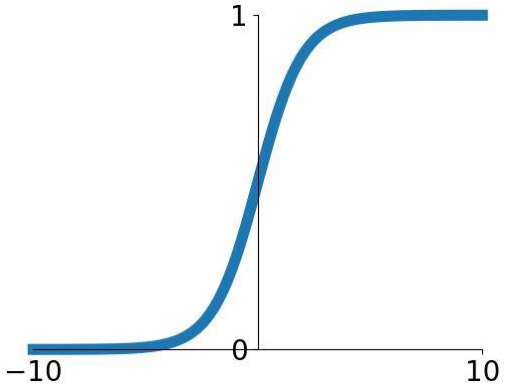
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# Activation Functions

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

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron



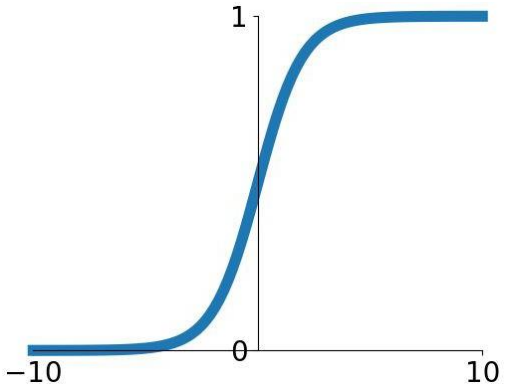

**Sigmoid**

1. Saturated neurons “kill” the gradients

# Activation Functions

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

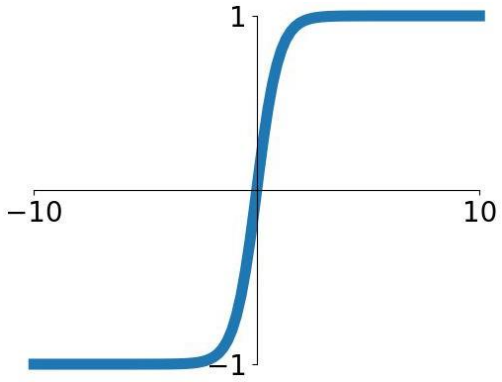
- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron




**Sigmoid**

1. Saturated neurons “kill” the gradients
2. exp() is a bit expensive

# Activation Functions

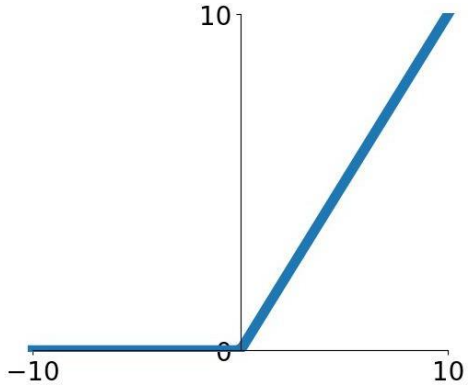



**tanh(x)**

- Squashes numbers to range [-1,1]
- still kills gradients when saturated :(

[LeCun et al., 1991]

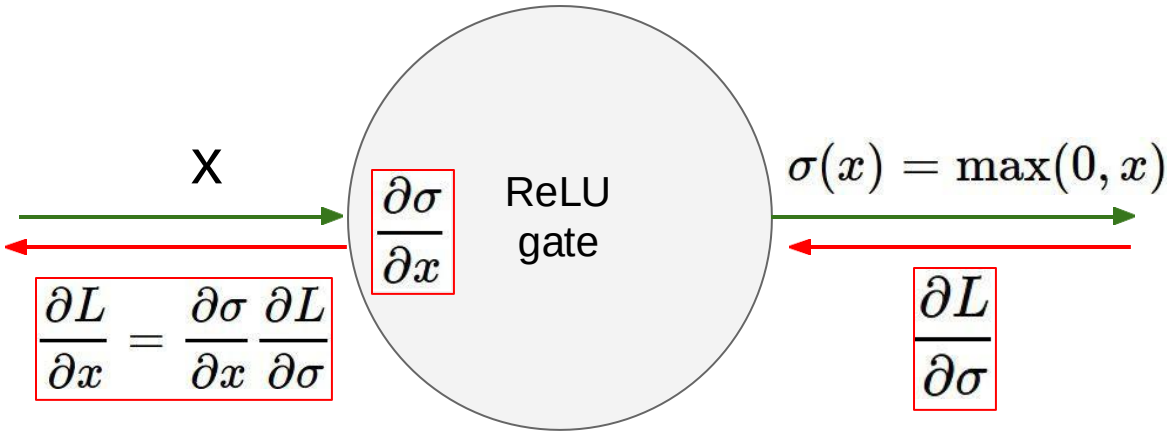
# Activation Functions

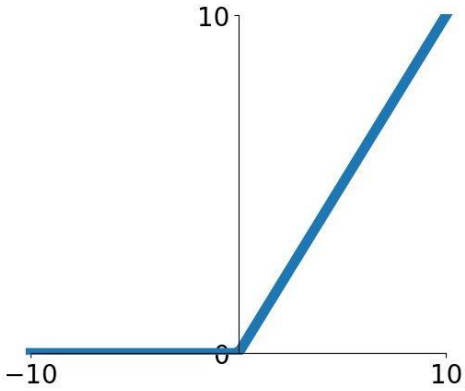



**ReLU**
(Rectified Linear Unit)

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)

[Krizhevsky et al., 2012]

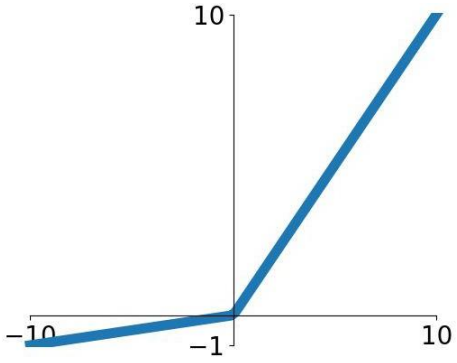



What happens when x = -10?
What happens when x = 0?
What happens when x = 10?

# Activation Functions

[Mass et al., 2013]
[He et al., 2015]



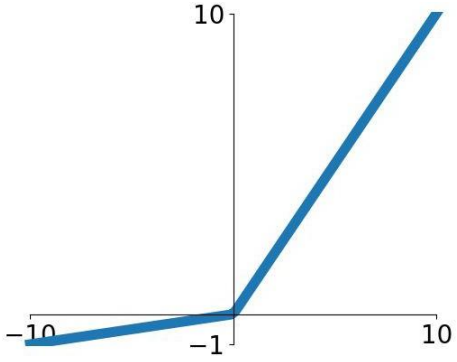

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

**Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

# Activation Functions

[Mass et al., 2013]
[He et al., 2015]




**Leaky ReLU**

$$f(x) = \max(0.01x, x)$$

- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not “die”.**

**Parametric Rectifier (PReLU)**

$$f(x) = \max(\alpha x, x)$$

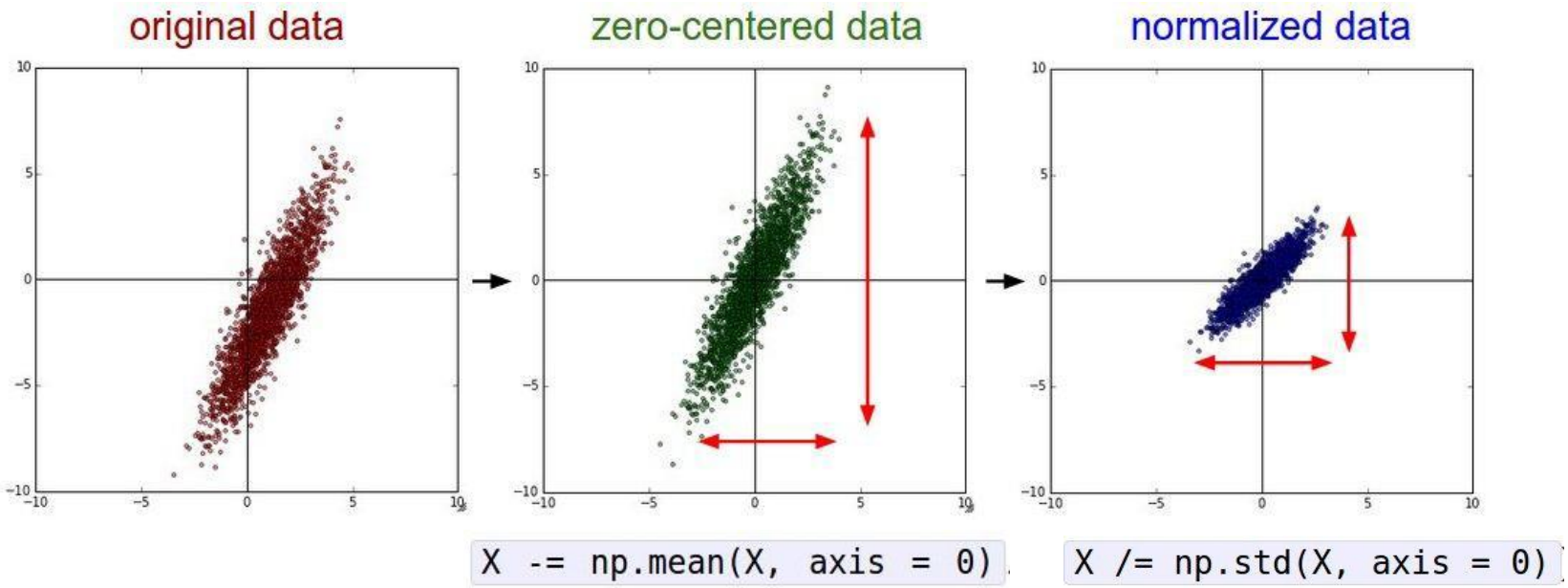
backprop into \alpha (parameter)

**TLDR: In practice:**

- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU
  - To squeeze out some marginal gains
- Don’t use sigmoid or tanh

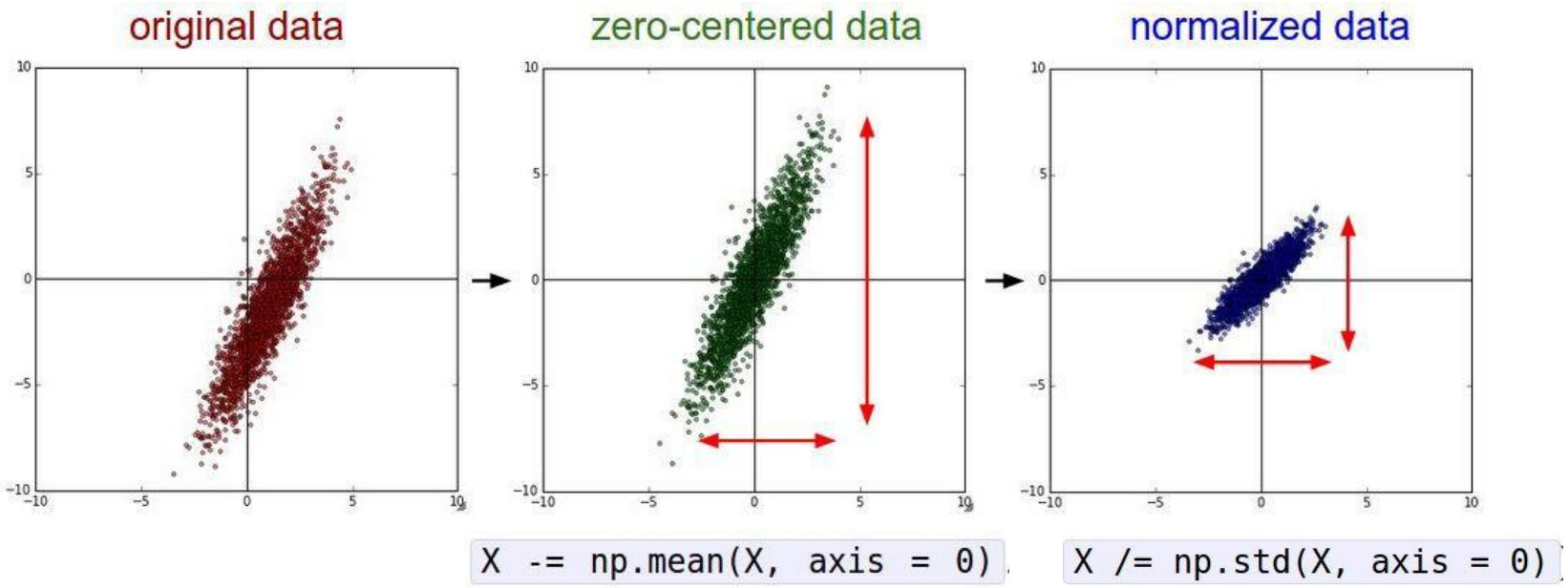
# Data Preprocessing

# Data Preprocessing




(Assume X [NxD] is data matrix, each example in a row)

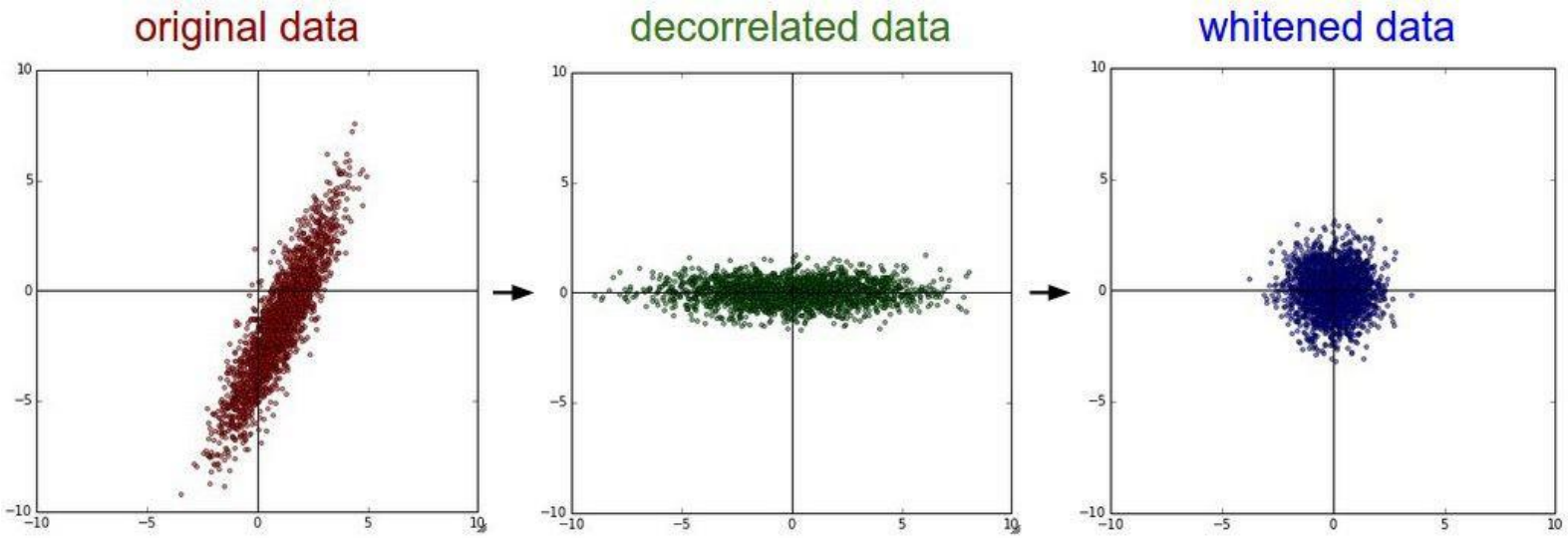
# Data Preprocessing




(Assume X [NxD] is data matrix, each example in a row)

# Data Preprocessing

In practice, you may also see **PCA** and **Whitening** of the data

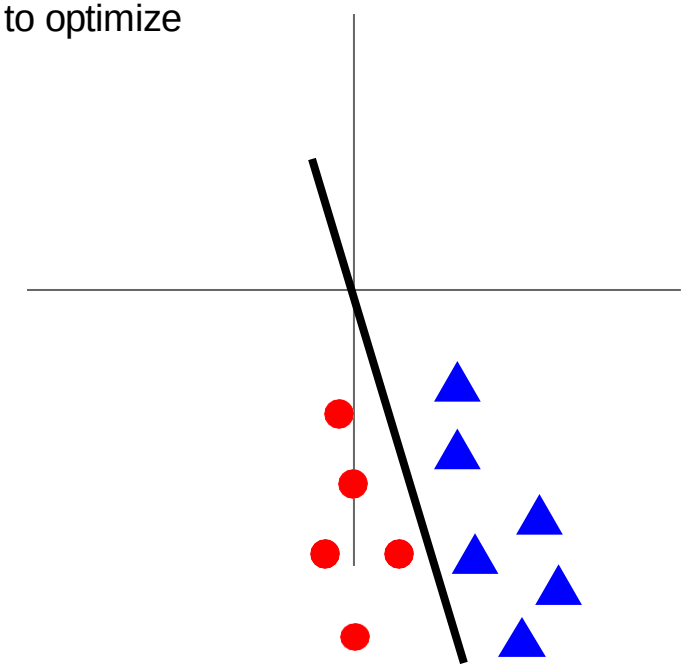



(data has diagonal covariance matrix)

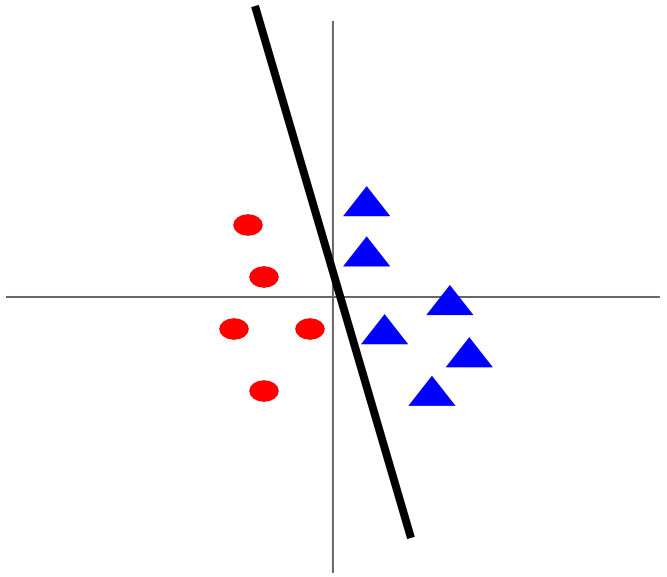
(covariance matrix is the identity matrix)

# Data Preprocessing

**Before normalization**: classification loss very sensitive to changes in weight matrix; hard to optimize



**After normalization**: less sensitive to small changes in weights; easier to optimize



**TLDR: In practice for Images:** center only

e.g. consider CIFAR-10 example with [32,32,3] images

- Subtract the mean image (e.g. AlexNet)
  (mean image = [32,32,3] array)
- Subtract per-channel mean (e.g. VGGNet)
  (mean along each channel = 3 numbers)
- Subtract per-channel mean and
  Divide by per-channel std (e.g. ResNet)
  (mean along each channel = 3 numbers)

Not common to do PCA or whitening

# Batch Normalization

# Batch Normalization

[Ioffe and Szegedy, 2015]

"you want zero-mean unit-variance activations? just make them so."

consider a batch of activations at some layer. To make each dimension zero-mean unit-variance, apply:

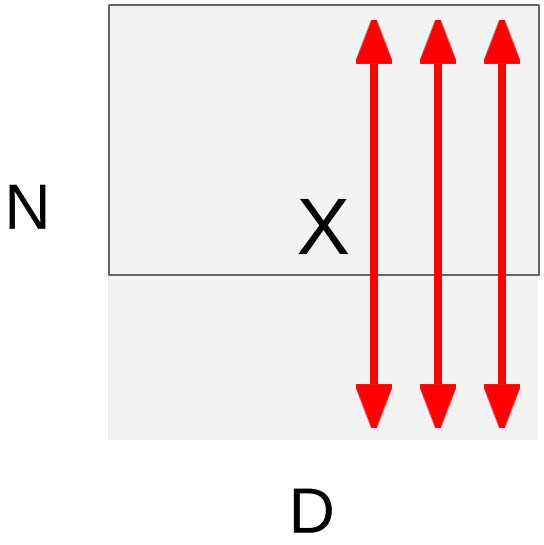
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** $x : N \times D$




$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

# Batch Normalization

[Ioffe and Szegedy, 2015]

**Input:** $x : N \times D$

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

# Batch Normalization: Test-Time

Estimates depend on minibatch; can't do this at test-time!

**Input:** $x : N \times D$

**Learnable scale and shift parameters:**

$$\gamma, \beta : D$$

Learning $\gamma = \sigma$, $\beta = \mu$ will recover the identity function!

$$\mu_j = \frac{1}{N} \sum_{i=1}^{N} x_{i,j}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^{N} (x_{i,j} - \mu_j)^2$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

# Batch Normalization: Test-Time

**Input:** $x : N \times D$

**Learnable scale and shift parameters:**

$\gamma, \beta : D$

During testing batchnorm becomes a linear operator!
Can be fused with the previous fully-connected or conv layer

$$\mu_j = \text{(Running) average of values seen during training}$$

Per-channel mean, shape is D

$$\sigma_j^2 = \text{(Running) average of values seen during training}$$

Per-channel var, shape is D

$$\hat{x}_{i,j} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \varepsilon}}$$

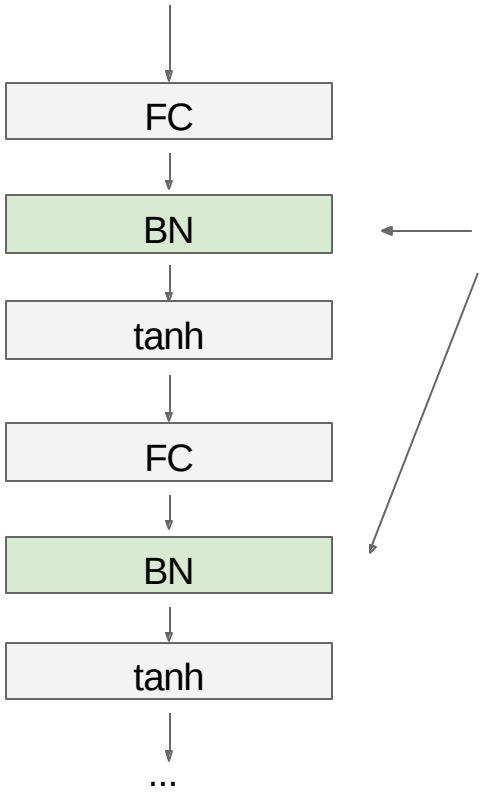
Normalized x, Shape is N x D

$$y_{i,j} = \gamma_j \hat{x}_{i,j} + \beta_j$$

Output, Shape is N x D

# Batch Normalization

[Ioffe and Szegedy, 2015]

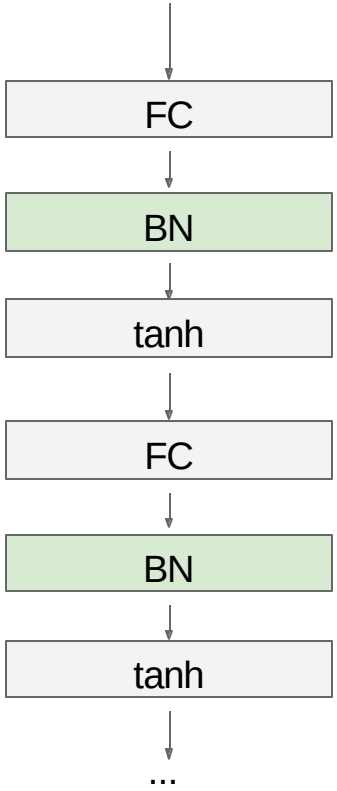



Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

# Batch Normalization

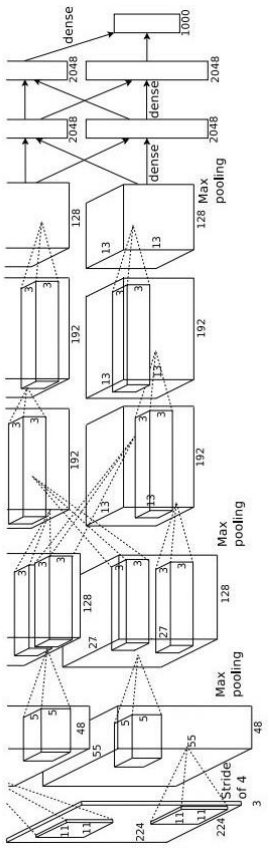
[Ioffe and Szegedy, 2015]



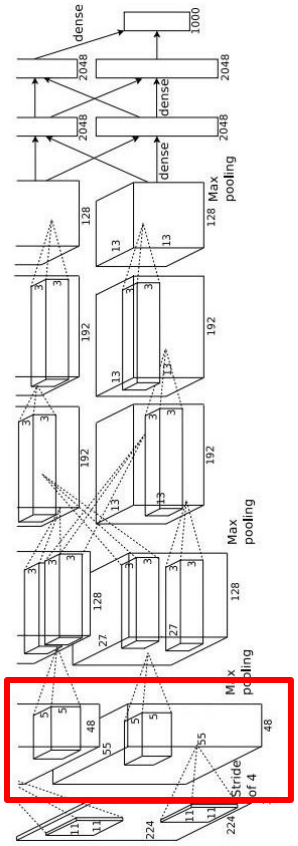

- Makes deep networks **much** easier to train!
- Improves gradient flow
- Allows higher learning rates, faster convergence
- Networks become more robust to initialization
- Acts as regularization during training
- Zero overhead at test-time: can be fused with conv!
- Behaves differently during training and testing: this is a very common source of bugs!

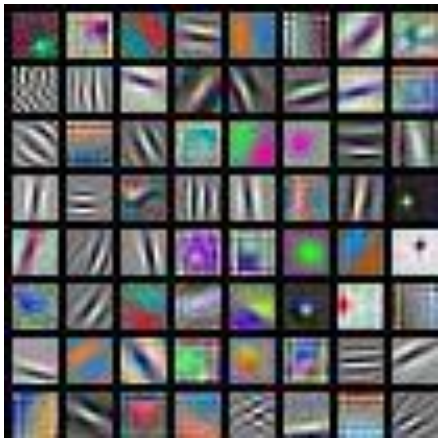
# Transfer learning

# Transfer Learning with CNNs



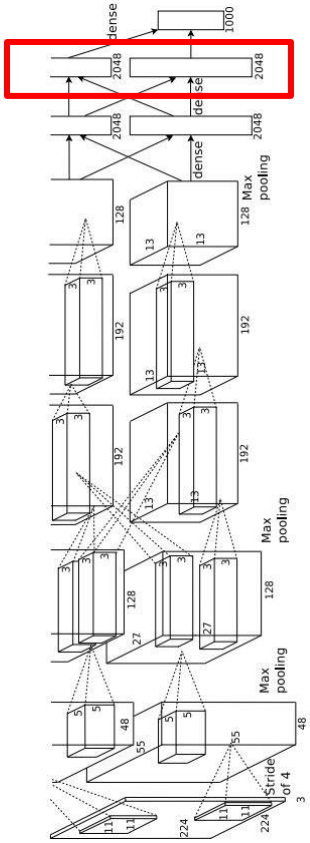
# Transfer Learning with CNNs



AlexNet:
64 x 3 x 11 x 11

# Transfer Learning with CNNs

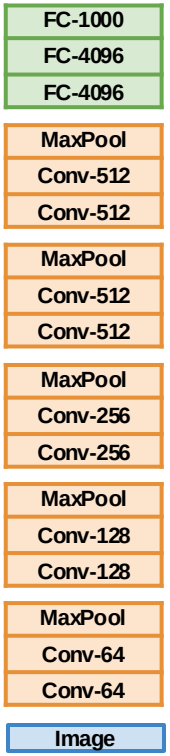



Test image L2 Nearest neighbors in <u>feature</u> space

# Transfer Learning with CNNs

Donahue et al, “DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition”, ICML 2014
Razavian et al, “CNN Features Off-the-Shelf: An Astounding Baseline for Recognition”, CVPR Workshops 2014

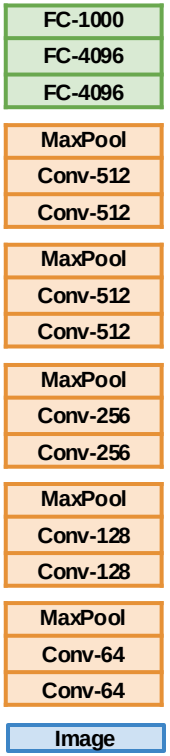
1. Train on Imagenet



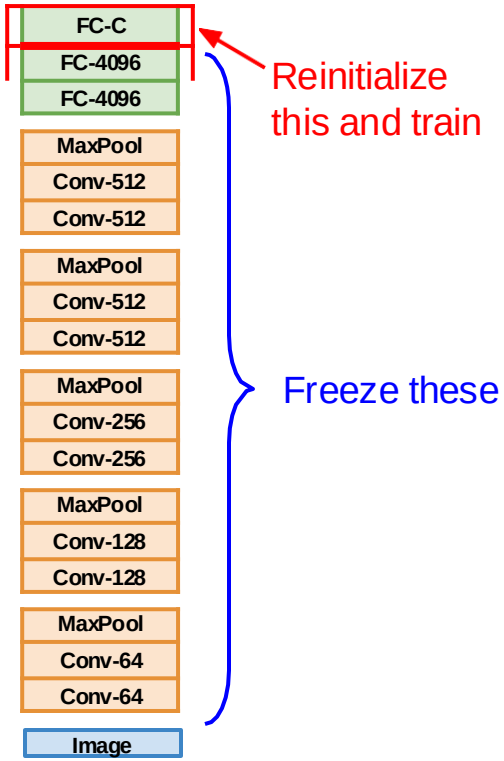
# Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet



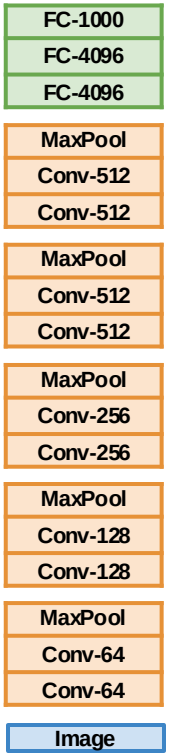

2. Small Dataset (C classes)



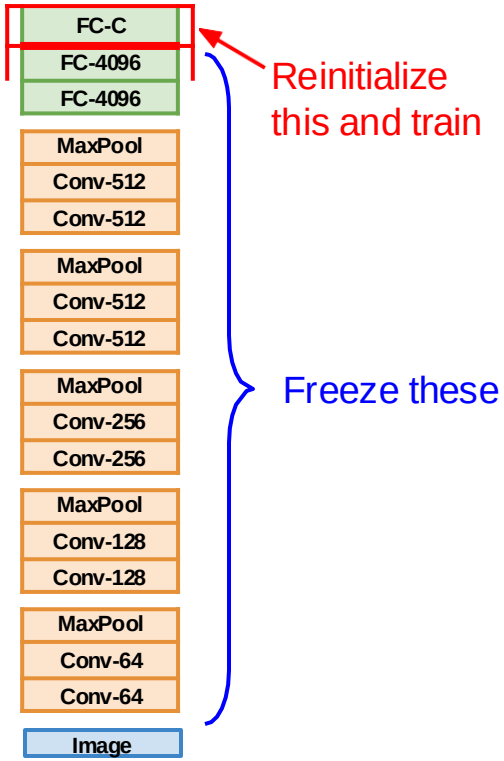
# Transfer Learning with CNNs

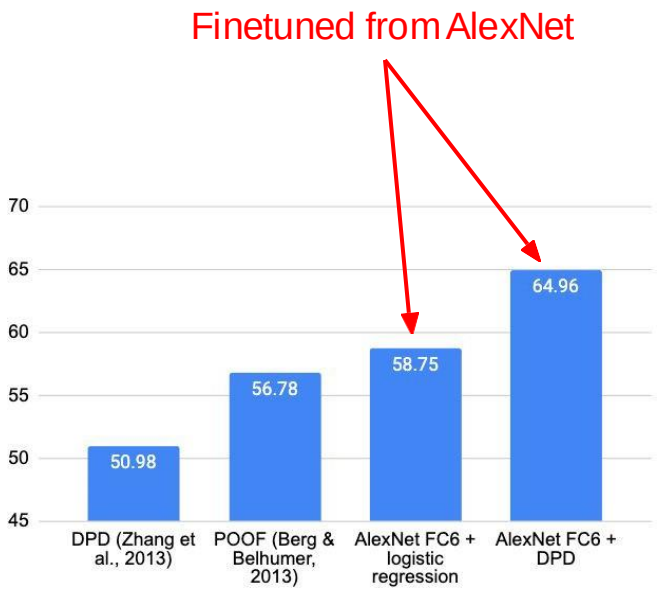
Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

1. Train on Imagenet




2. Small Dataset (C classes)



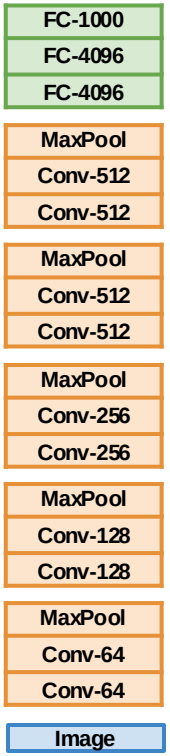




Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014

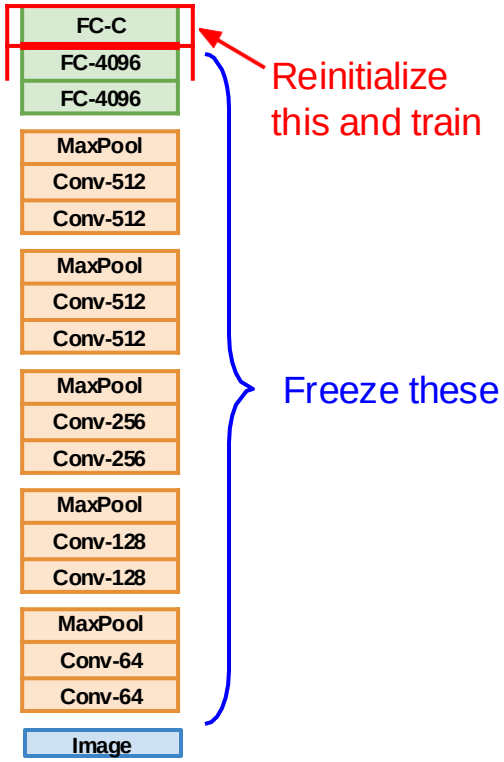
# Transfer Learning with CNNs

Donahue et al, "DeCAF: A Deep Convolutional Activation Feature for Generic Visual Recognition", ICML 2014
Razavian et al, "CNN Features Off-the-Shelf: An Astounding Baseline for Recognition", CVPR Workshops 2014

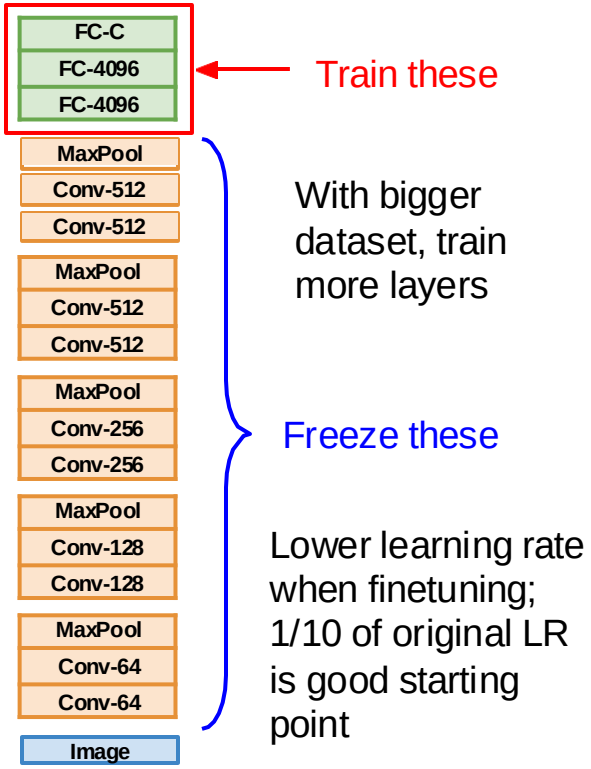
1. Train on Imagenet




2. Small Dataset (C classes)




3. Bigger dataset



Summary
We looked in detail at:

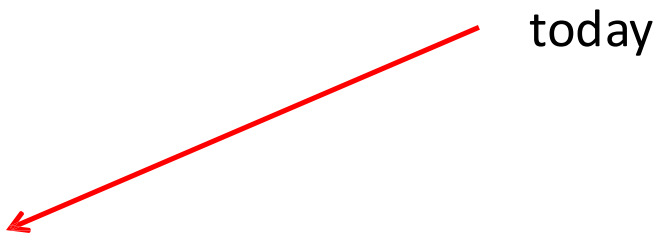
TLDRs

- Activation Functions (use ReLU)
- Data Preprocessing (images: subtract mean)
- Batch Normalization (use this!)
- Transfer learning (use this if you can!)


# Optimization

# Optimization is essential for DL

- Deep Learning is an instance of a recipe:
  1. Specification of a dataset
  2. A cost function
  3. A model
  4. An optimization procedure

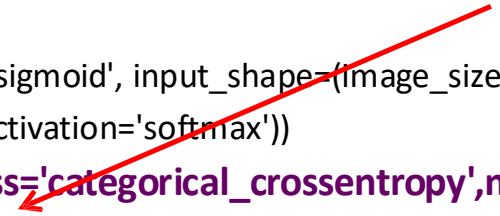


# Our focus is on one case of optimization

- Find parameters $\boldsymbol{\theta}$ of a neural network that significantly reduces a cost function $J(\boldsymbol{\theta})$
  - It typically includes:
    - a performance measure evaluated on an entire training set as well as an additional regularization term

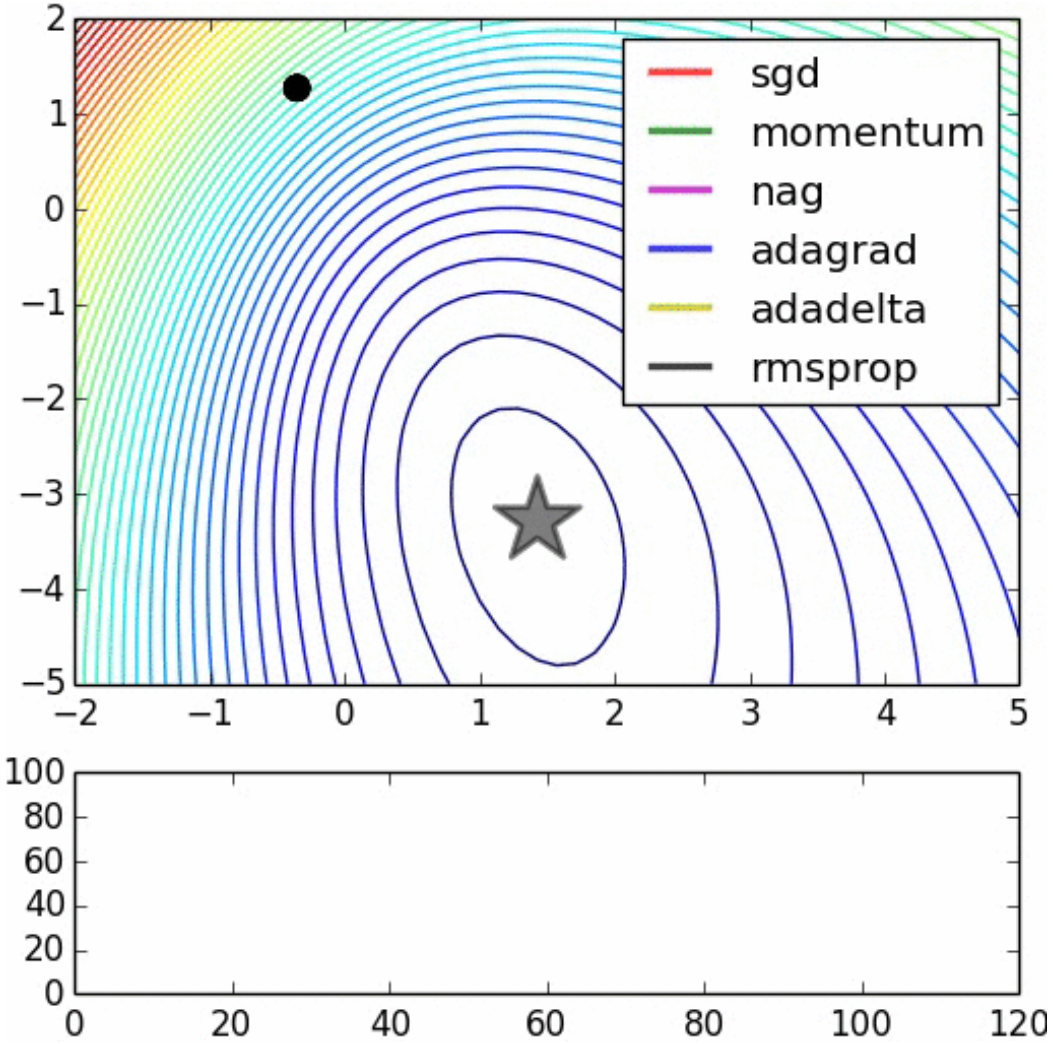
# Keras for MNIST Neural Network

- # Neural Network
- import keras
- from keras.datasets import mnist
- from keras.layers import Dense
- from keras.models import Sequential
- (x_train, y_train), (x_test, y_test) = mnist.load_data()
- num_classes=10
- image_vector_size=28*28
- x_train = x_train.reshape(x_train.shape[0], image_vector_size)
- x_test = x_test.reshape(x_test.shape[0], image_vector_size)
- y_train = keras.utils.to_categorical(y_train, num_classes)
- y_test = keras.utils.to_categorical(y_test, num_classes)
- image_size = 784 model = Sequential()
- model.add(Dense(units=32, activation='sigmoid', input_shape=(image_size,)))
- model.add(Dense(units=num_classes, activation='softmax'))
- **model.compile(optimizer='sgd', loss='categorical_crossentropy',metrics=['accuracy'])**
- history = model.fit(x_train, y_train, batch_size=128, epochs=10, verbose=False,validation_split=.1)
- loss,accuracy = model.evaluate(x_test, y_test, verbose=False)



UCMERCED

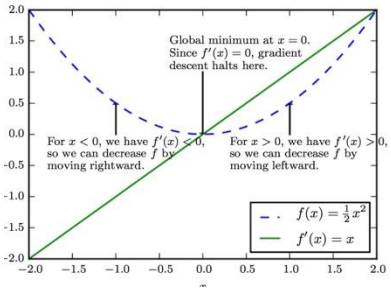
# Optimization methods

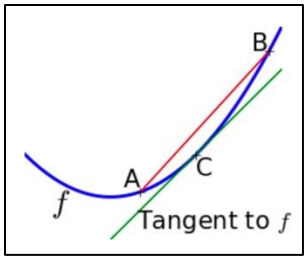



http://hduongtrong.github.io/2015/11/23/coordinate-descent/

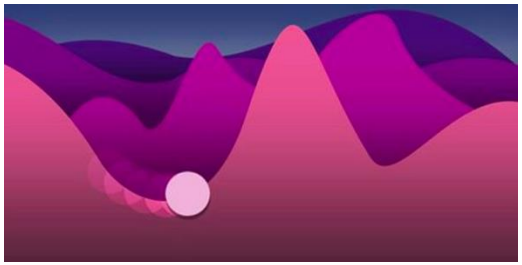
# Optimization Problem in DL

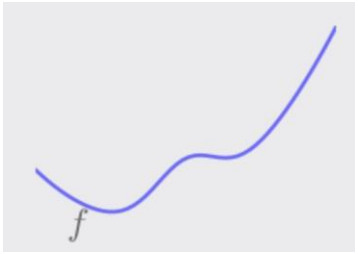
- Optimization is an extremely difficult task for DL
  - Traditional ML: careful design of objective function and constraints to ensure convex optimization




  - When training neural networks, we must confront nonconvex cases






UCMERCED

# Batch Gradient Methods

- Batch or deterministic gradient methods:
  - Optimization methods that use all training samples are batch or deterministic methods

- *Somewhat confusing terminology*
  - Batch also used to describe *minibatch* used by minibatch stochastic gradient descent
  - Batch gradient descent implies use of full training set
  - Batch size refers the size of a minibatch

# Stochastic or Online Methods

- Those using a single sample are called Stochastic or on-line
  - On-line typically means continually created samples drawn from a stream rather than multiple passes over a fixed size training set
- Deep learning algorithms usually use more than one but fewer than all samples
  - Methods traditionally called minibatch or minibatch stochastic now simply called stochastic

Ex: (stochastic gradient descent - SGD)

# Minibatch Size

- Driven by following:
  - Larger batches→more accurate gradient
  - If all examples processed in parallel, amount of memory scales with batch size
    - This is a limiting factor in batch size
  - GPU architectures more efficient with sizes power of 2
    - Range from 32 to 256, sometimes with 16 for large models

# SGD and Generalization Error

- Minibatch SGD follows the gradient of the true generalization error

$$J^*(\boldsymbol{\theta}) = E_{(\boldsymbol{x},y)\sim p_{data}} \left( L(f(\boldsymbol{x};\boldsymbol{\theta}), y) \right)$$

  - as long as the examples are repeated
- Implementations of minibatch SGD
  - Shuffle once and pass through multiple number of times

# SGD Follows Gradient Estimate Downhill

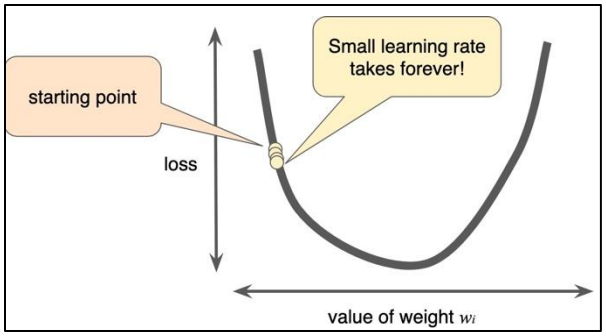
**Algorithm:** SGD update at training iteration $k$

**Require:** Learning rate $\epsilon_k$.
**Require:** Initial parameter $\boldsymbol{\theta}$
  **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: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
  **end while**

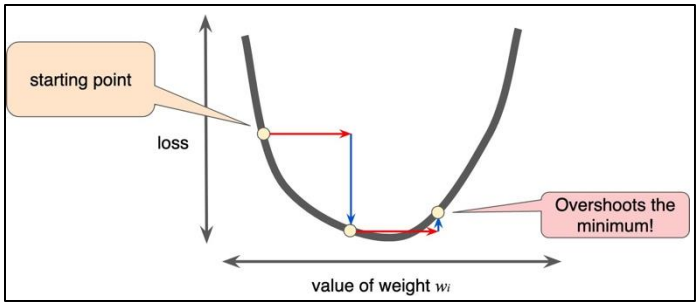
A crucial parameter is the learning rate $\varepsilon$

At iteration $k$ it is $\varepsilon_k$

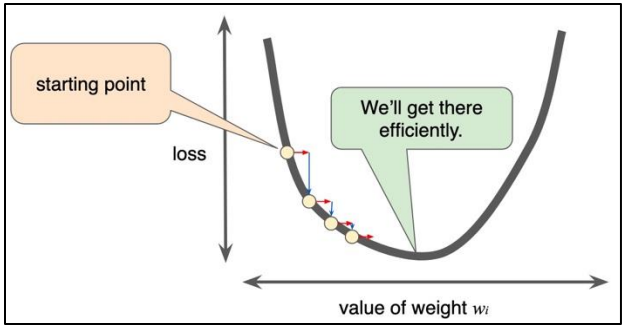
# Choice of Learning Rate




Too small learning rate will take too long




Too large, the next point will perpetually bounce haphazardly across the bottom of the well




If gradient is small, then can safely try a larger learning rate, which compensates for the small gradient and results in a larger step size

https://developers.google.com/machine-learning/crash-course/reducing-loss/learning-rate

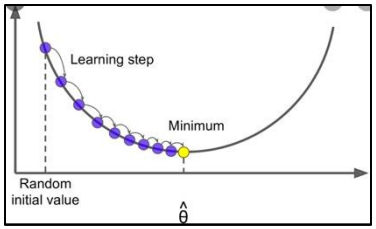
# Need for Decreasing Learning Rate

- True gradient of total cost function
  - Becomes small and then $0$
    - One can use a fixed learning rate
- But SGD has a source of noise
  - Random sampling of $m$ training samples
    - Gradient does not vanish even when arriving at a minimum
  - Common to decay learning rate linearly until iteration $\tau$: $\varepsilon_k=(1-\alpha)\varepsilon_0+\alpha\varepsilon_\tau$ with $\alpha=k/\tau$
  - After iteration $\tau$, it is common to leave $\varepsilon$ constant
    - Often a small positive value in the range $0.0$ to $1.0$

# Learning Rate Decay

- Decay learning rate

  $\tau$: $\varepsilon_k=(1-\alpha)\varepsilon_0+\alpha\varepsilon_\tau$ with $\alpha=k/\tau$




- Learning rate is calculated at each update
  - (e.g. end of each mini-batch) as follows:

```
lrate = initial_lrate * (1 / (1 + decay * iteration))
```

  - Where *lrate* is learning rate for current epoch
  - *initial_lrate* is specified as an argument to SGD
  - *decay* is the decay rate which is greater than zero and
  - *iteration* is the current update number

```
from keras.optimizers import SGD
...
opt = SGD(lr=0.01, momentum=0.9, decay=0.01)
model.compile(..., optimizer=opt)
```

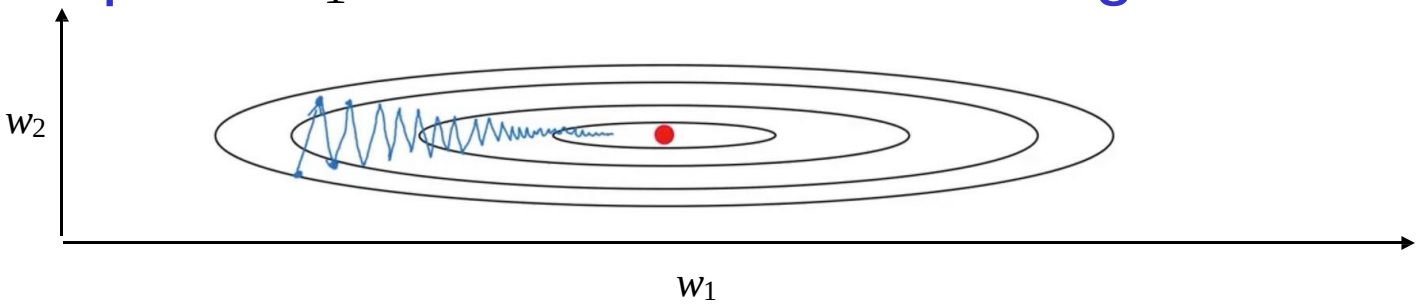
UCMERCED

# Momentum Method

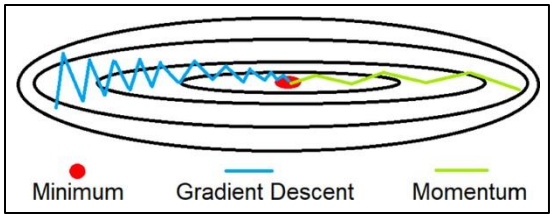
- SGD is a popular optimization strategy but it can be slow
- Momentum method accelerates learning, when:
  - Facing high curvature
  - Noisy gradients
- It works by accumulating the moving average of past gradients and moves in that direction while exponentially decaying

# Gradient Descent with Momentum

- Gradient descent with momentum converges faster than standard gradient descent
- Taking large steps in $w_2$ direction and small steps in $w_1$ direction slows down algorithm




- Momentum reduces oscillation in $w_2$ direction




- Now can set a higher learning rate

https://www.andreaperlato.com/aipost/gradient-descent-with-momentum/

# Momentum Definition

- Introduce velocity variable $\boldsymbol{v}$
- This is the direction and speed at which parameters move through parameter space
- Name momentum comes from physics & is mass times velocity
  - The momentum algorithm assumes unit mass
- A hyperparameter $\alpha \in [0,1)$ determines exponential decay of $\boldsymbol{v}$

# Momentum Update Rule

- The update rule is given by

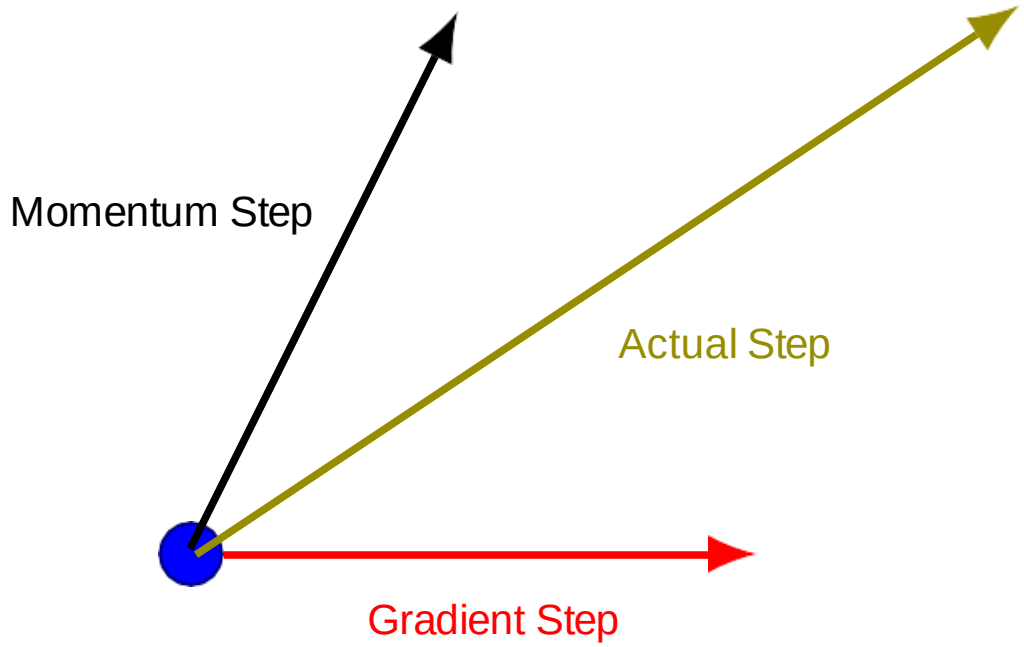
$$v \leftarrow \alpha v - \varepsilon \nabla_\theta \left( \frac{1}{m} \sum_{i=1}^{m} L\left( f(x^{(i)}; \theta), y^{(i)} \right) \right)$$
$$\theta \leftarrow \theta + v$$

- The velocity $\boldsymbol{v}$ accumulates the gradient

$$\nabla_\theta \left( \frac{1}{m} \sum_{i=1}^{m} L\left( f(x^{(i)}; \theta), y^{(i)} \right) \right)$$

  elements
- The larger $\alpha$ is relative to $\varepsilon$, the more previous gradients affect the current direction
- The SGD algorithm with momentum is next

# Momentum



# SGD Algorithm with Momentum

## Algorithm: SGD with momentum

**Require:** Learning rate $\epsilon$, momentum parameter $\alpha$.
**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 targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
**end while**

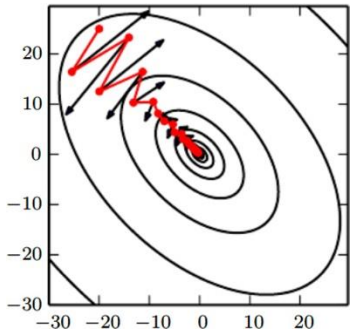
Keras: The learning rate can be specified via the *lr* argument and the momentum can be specified via the *momentum* argument.

```
1 from keras.optimizers import SGD
2 ...
3 opt = SGD(lr=0.01, momentum=0.9)
4 model.compile(..., optimizer=opt)
```

UCMERCED

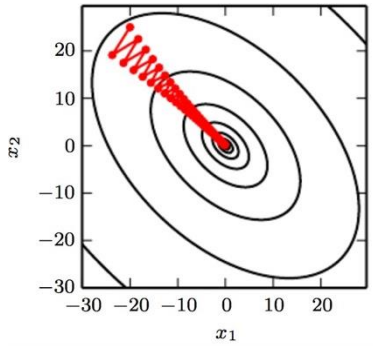
# Momentum

- SGD with momentum




Contour lines depict a quadratic loss function with a poorly conditioned Hessian matrix.
Red path cutting across the contours depicts path followed by momentum learning rule as it minimizes this function

- Comparison to SGD without momentum

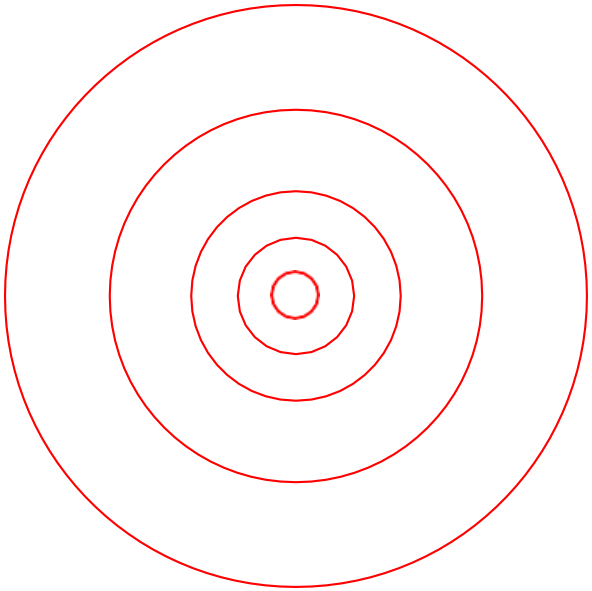



At each step we show path that would be taken by SGD at that step
Poorly conditioned quadratic objective
Looks like a long narrow valley with steep sides
Wastes time

# Importance of Learning Rate

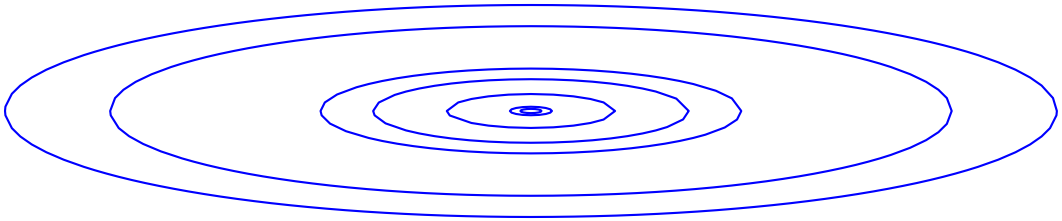
- Learning rate is the most difficult hyperparameter to set
  - It significantly affects model performance
- Cost is highly sensitive to some directions in parameter space and insensitive to others
  - Momentum helps but introduces another hyperparameter
  - Other approach
    - If direction of sensitivity is axis aligned, have a separate learning rate for each parameter and adjust them throughput learning

# Motivation



Nice (all features are equally important)

# Motivation



Harder

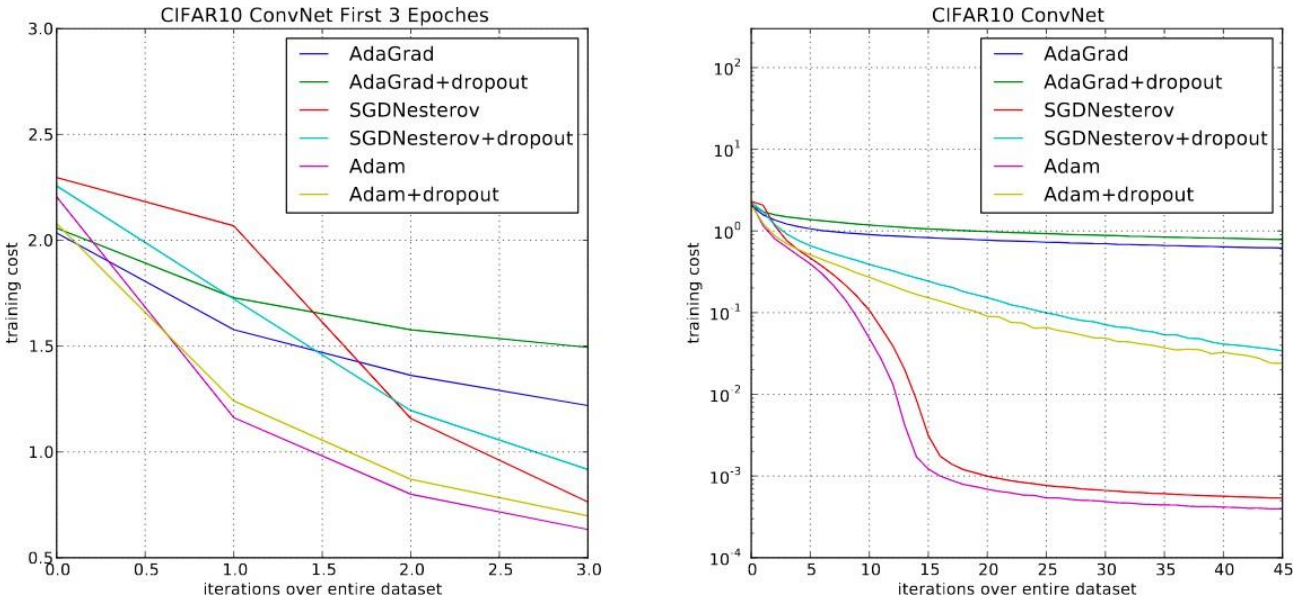
# AdaGrad

- Individually adapts learning rates of all parameters
  - Scale them inversely proportional to the sum of the historical squared values of the gradient
- The AdaGrad Algorithm:

**Require:** Global learning rate $\epsilon$
**Require:** Initial parameter $\boldsymbol{\theta}$
**Require:** Small constant $\delta$, perhaps $10^{-7}$, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
**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**

Performs well for some but not all deep learning

# Performance with CNN




Convolutional neural networks training cost.
(left) Training cost for the first three epochs.
(right) Training cost over 45 epochs.
CIFAR-10 with c64-c64-c128-1000 architecture.

UCMERCED