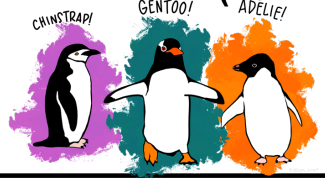
Deep Learning – Part 3

Road map

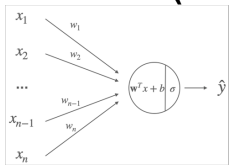
Lectures

1. Introduction

2. ML Basics (Part 1)

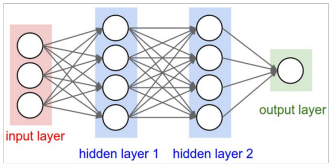


3. ML Basics (Part 2)

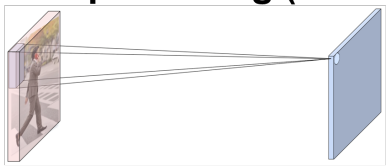


4. Role of Input

5. Deep Learning (Part 1)



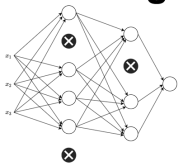
6. Deep Learning (Part 2)



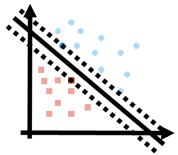
Road map

Lectures

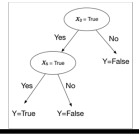
7. **Deep Learning (Part 3)**



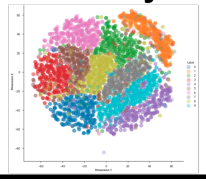
8. **Support Vector Machine**



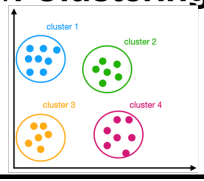
9. **Other Supervised Learning Models**



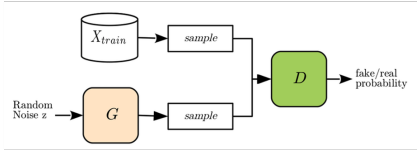
10. **Dimensionality Reduction**



11. **Clustering**



12. **Other Learning Examples**



So far

- Neural networks
  - Layers (fully connected or convolutional)
  - Activation functions
  - Loss functions

Useful links

- https://playground.tensorflow.org/

  https://poloclub.github.io/cnn-explainer/

  https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html

  https://www.cs.ryerson.ca/~aharley/vis/conv/

  https://teachablemachine.withgoogle.com/

  vitademo.epfl.ch/movements

Outline

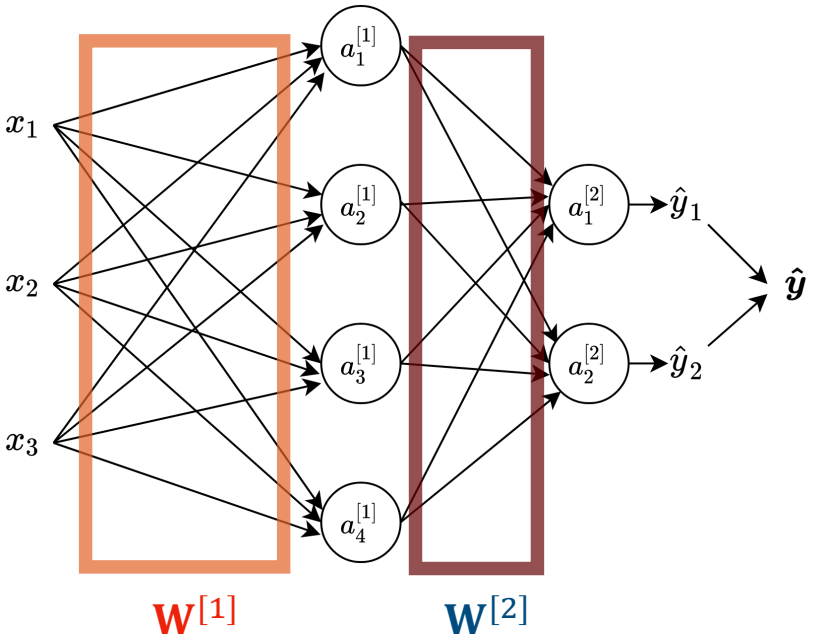
- Backprop
- Recipe for training neural networks
- Weight initialisation
- Optimization
- Batch normalization
- Regularisation techniques
- Transfer learning

- It s all about the gradient!

Training neural nets

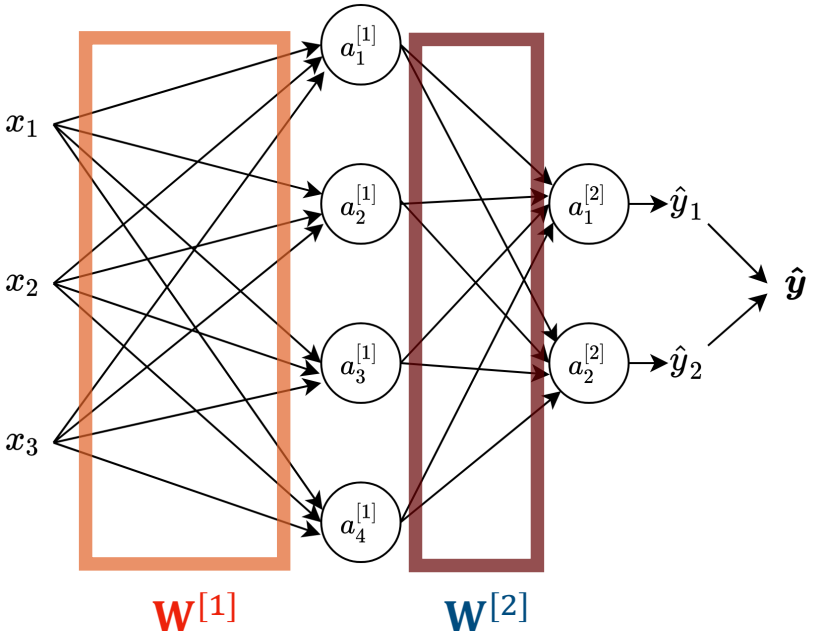
Neural networks
Training

- Forward pass of 2 layer NN (for a single sample):
  - $\mathbf{z}^{[1]} = \mathbf{W}^{[1]T}\boldsymbol{x} + \mathbf{b}^{[1]}$
  - $\mathbf{a}^{[1]} = g^{[1]}(\mathbf{z}^{[1]})$
  - $\mathbf{z}^{[2]} = \mathbf{W}^{[2]T}\mathbf{a}^{[1]} + \mathbf{b}^{[2]}$
  - $\hat{\mathbf{y}} = \mathbf{a}^{[2]} = g^{[2]}(\mathbf{z}^{[2]})$
  - $\hat{\mathbf{y}} = g^{[2]}(\mathbf{W}^{[2]T} g^{[1]}(\mathbf{W}^{[1]T}\boldsymbol{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]})$



Neural networks
Training

- Forward pass of 2 layer NN (for a single sample):
  - $\hat{\mathbf{y}} = g^{[2]}(\mathbf{W}^{[2]T} g^{[1]}(\mathbf{W}^{[1]T}\boldsymbol{x} + \mathbf{b}^{[1]}) + \mathbf{b}^{[2]})$
- To train, we need a **loss function**: $L(\hat{\mathbf{y}}, \mathbf{y})$
- Using that loss function, we want to update $\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \mathbf{W}^{[2]}, \mathbf{b}^{[2]}$
- using **gradient descent**.



Neural networks
Training

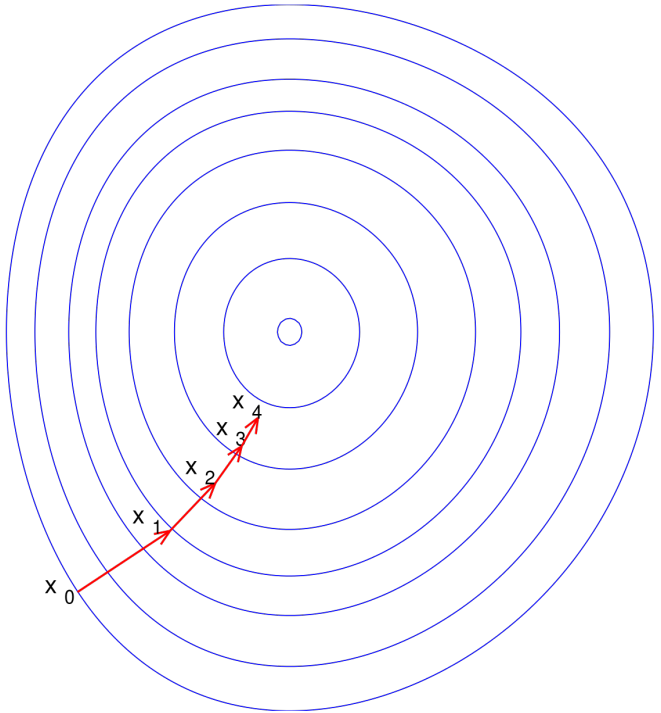
Need to compute: $\frac{\partial L}{\partial \mathbf{W}^{[i]}}, \frac{\partial L}{\partial \mathbf{b}^{[i]}}$

=> Gradient of loss with respect to weights

Once gradients are computed, update weights with:

- $\mathbf{W}^{[i]} := \mathbf{W}^{[i]} - \alpha \frac{\partial L}{\partial \mathbf{W}^{[i]}}$
- $\mathbf{b}^{[i]} := \mathbf{b}^{[i]} - \alpha \frac{\partial L}{\partial \mathbf{b}^{[i]}}$

where α is the learning rate



Backpropagation

NN - Backpropagation

Overview

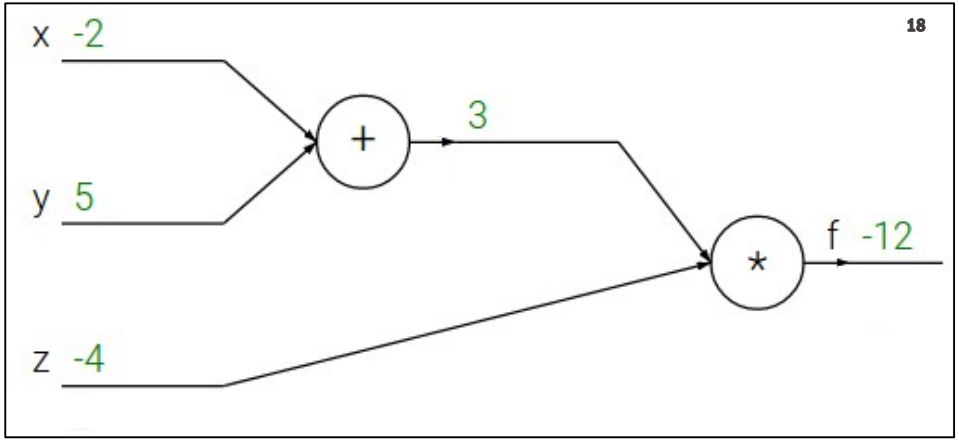
- Algorithm used to efficiently compute gradient of loss with respect to weights
- Makes use of chain rule: $\frac{\partial L}{\partial x} = \frac{dy}{dx}\frac{\partial L}{\partial y}$
- Use computational graph to progressively compute gradients

- **Forward pass**: Compute output
- **Backward pass**: Compute derivatives

Backprop

$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4



Backprop

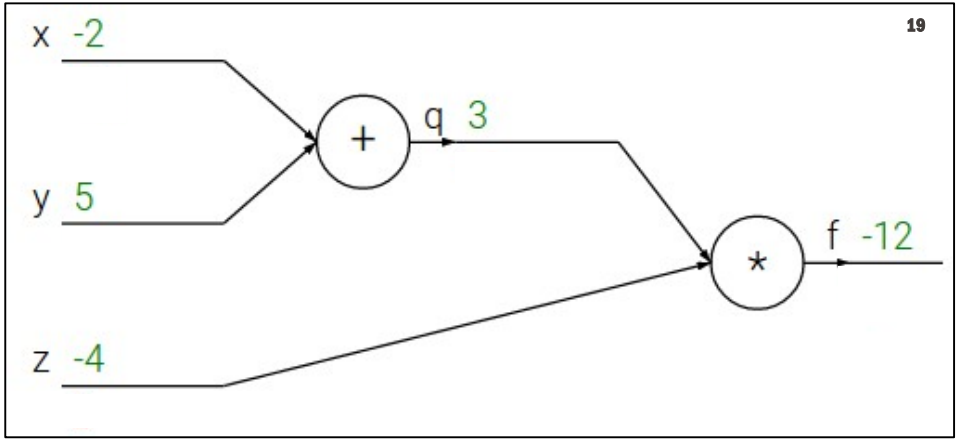
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

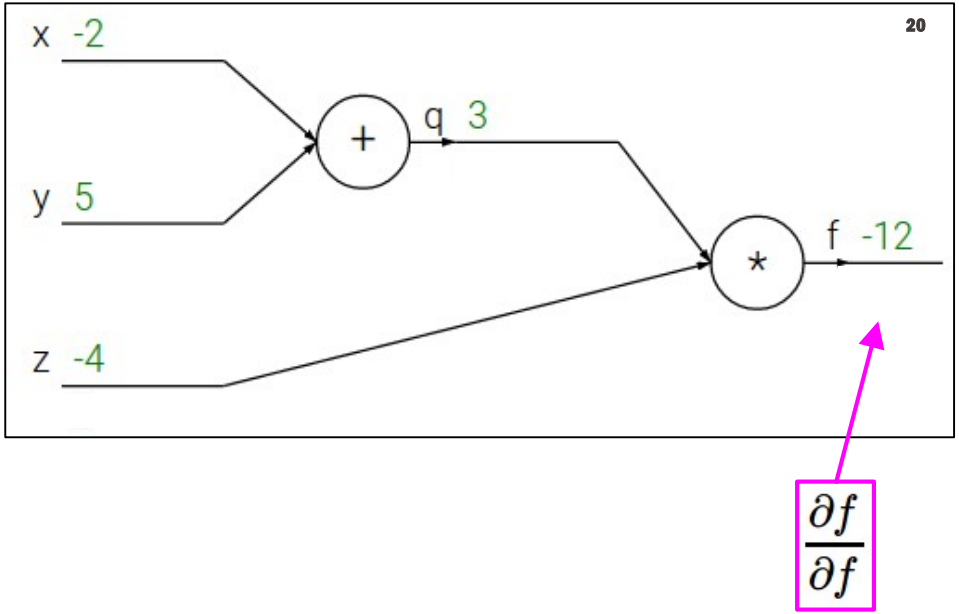
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

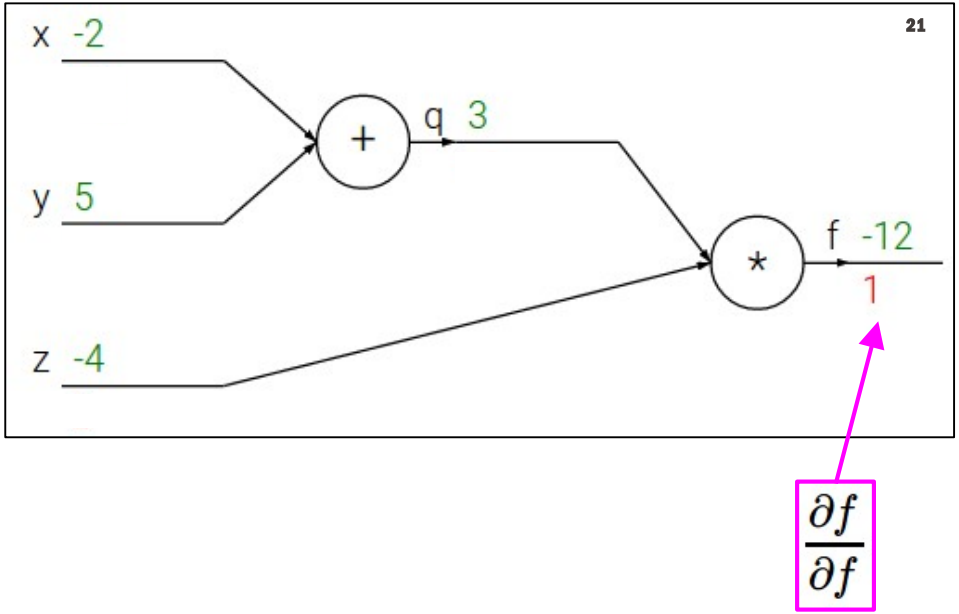
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

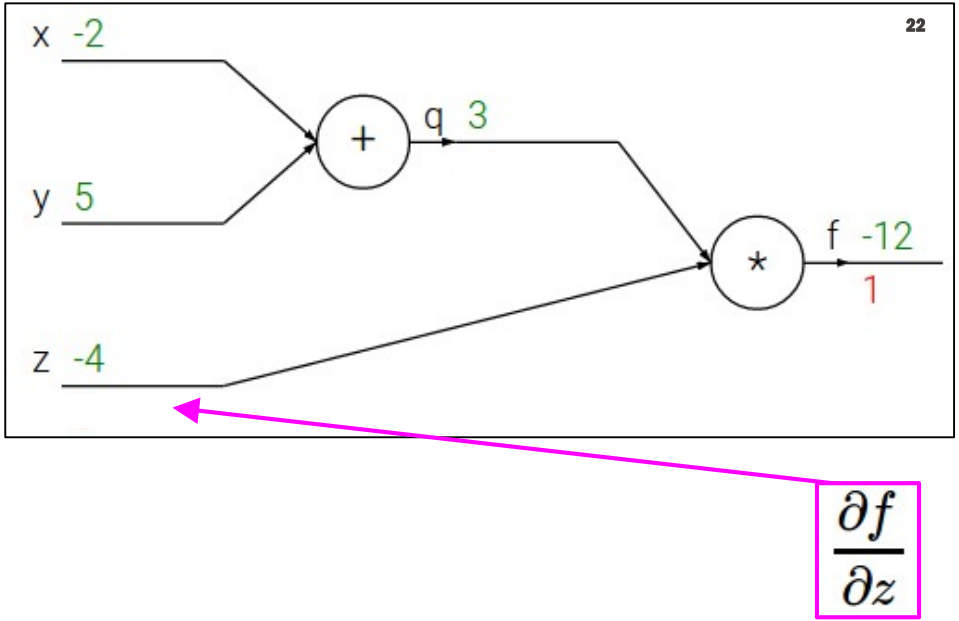
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

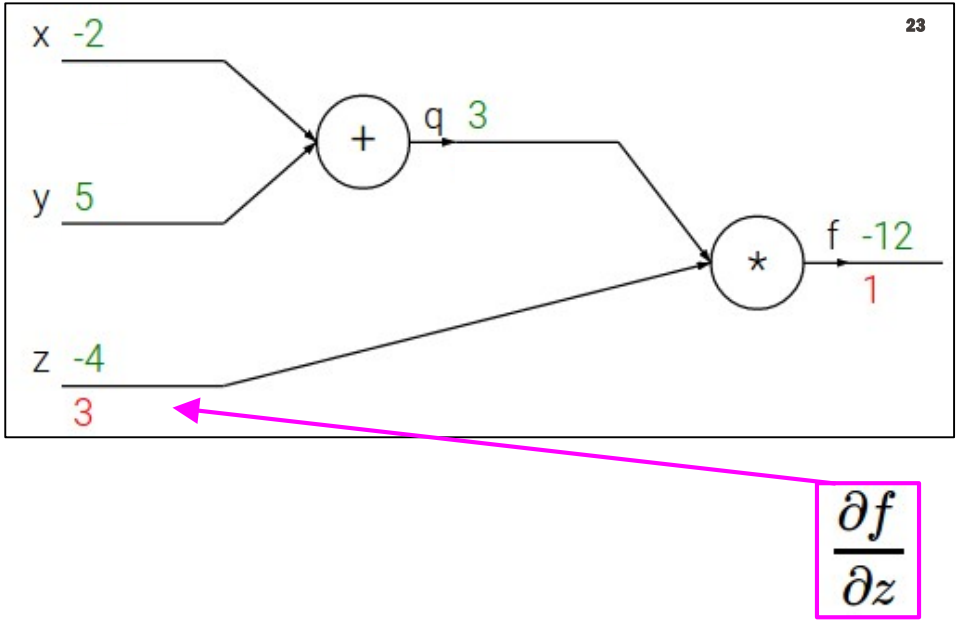
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

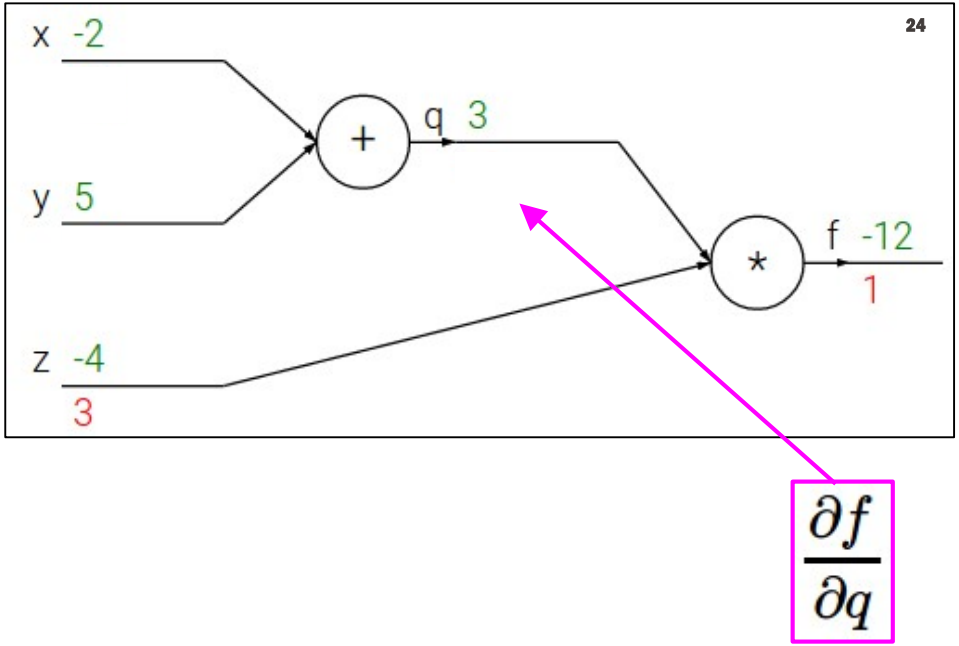
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

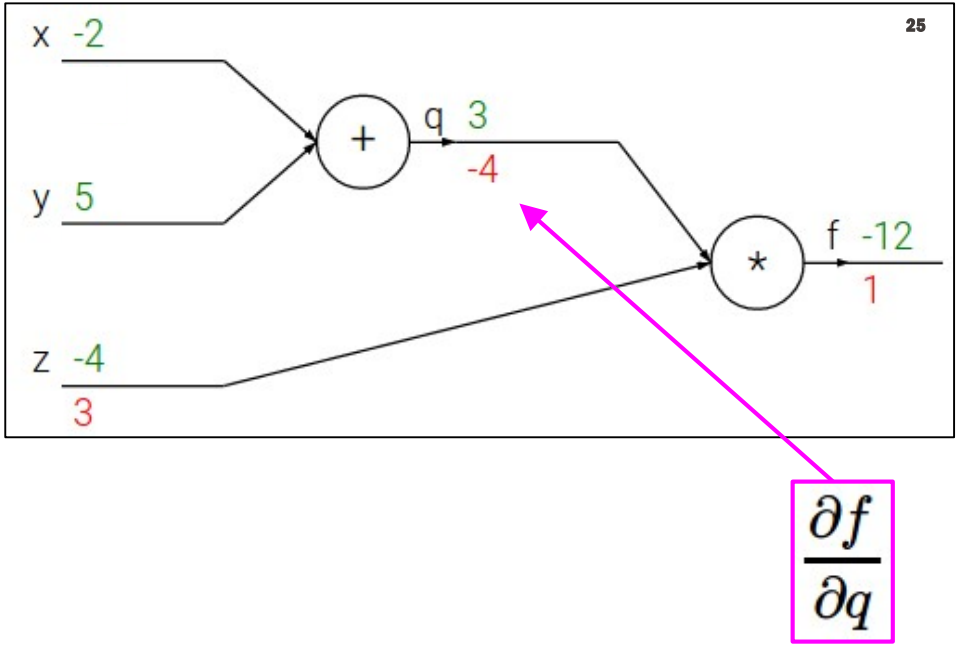
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

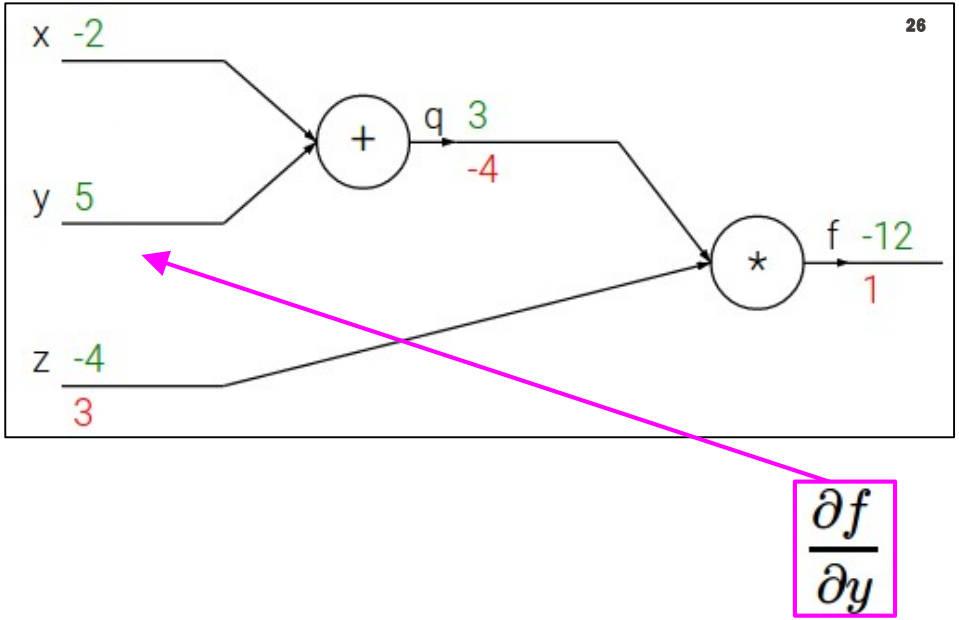
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

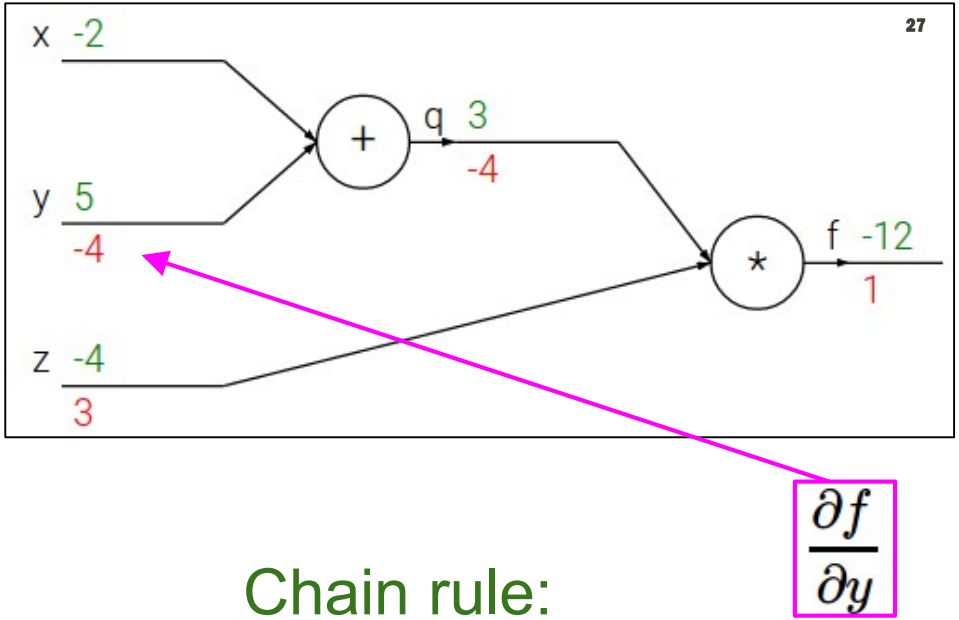
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Chain rule:

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y}$$

Backprop

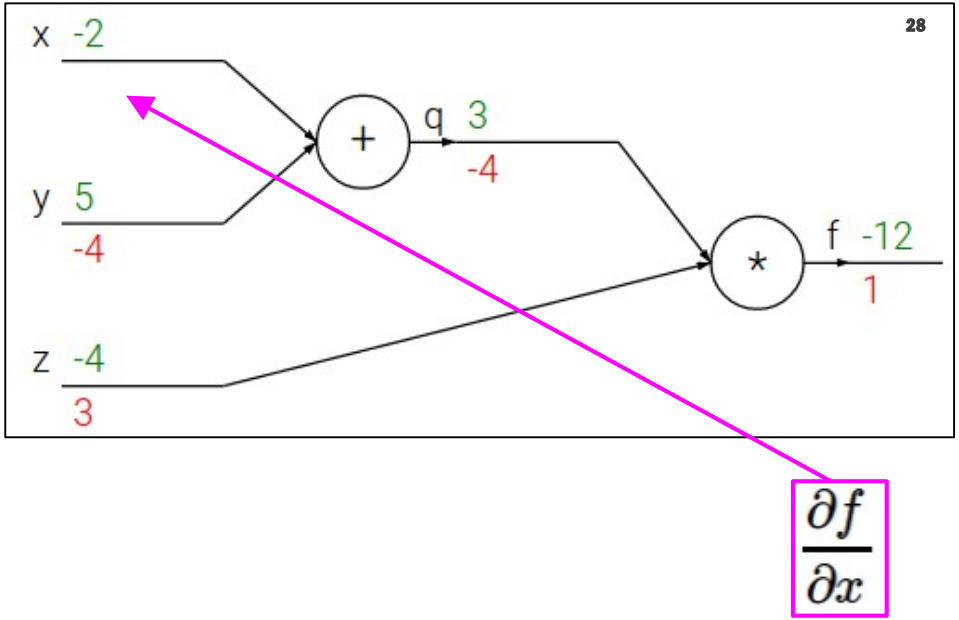
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$



Backprop

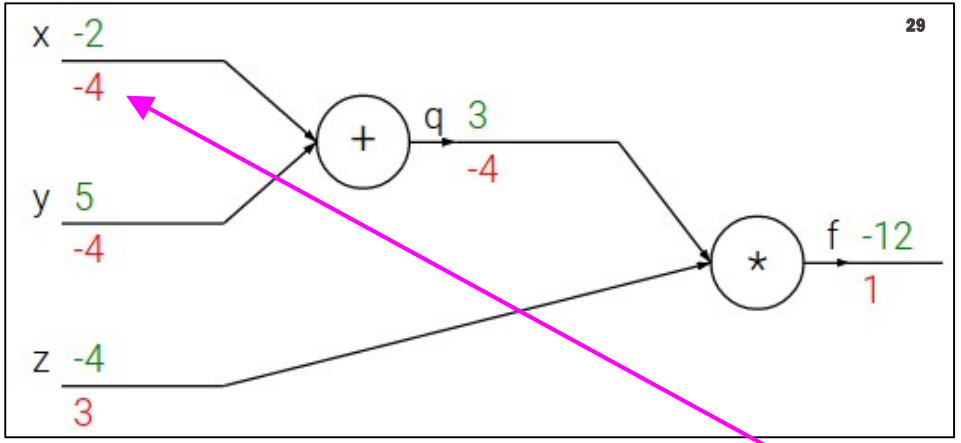
$$f(x, y, z) = (x + y)z$$

e.g., x = -2, y = 5, z = -4

$$q = x + y \qquad \frac{\partial q}{\partial x} = 1, \frac{\partial q}{\partial y} = 1$$

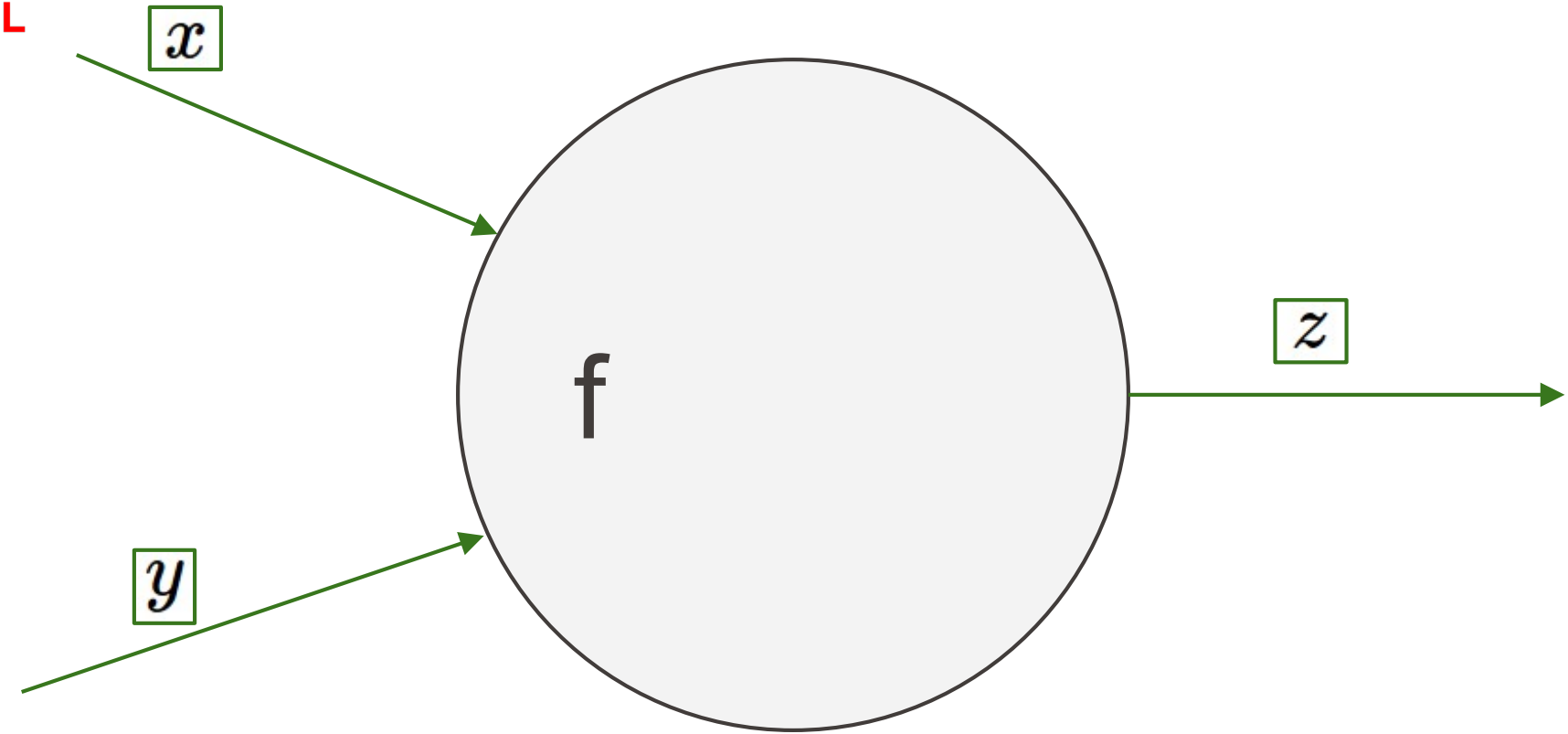
$$f = qz \qquad \frac{\partial f}{\partial q} = z, \frac{\partial f}{\partial z} = q$$

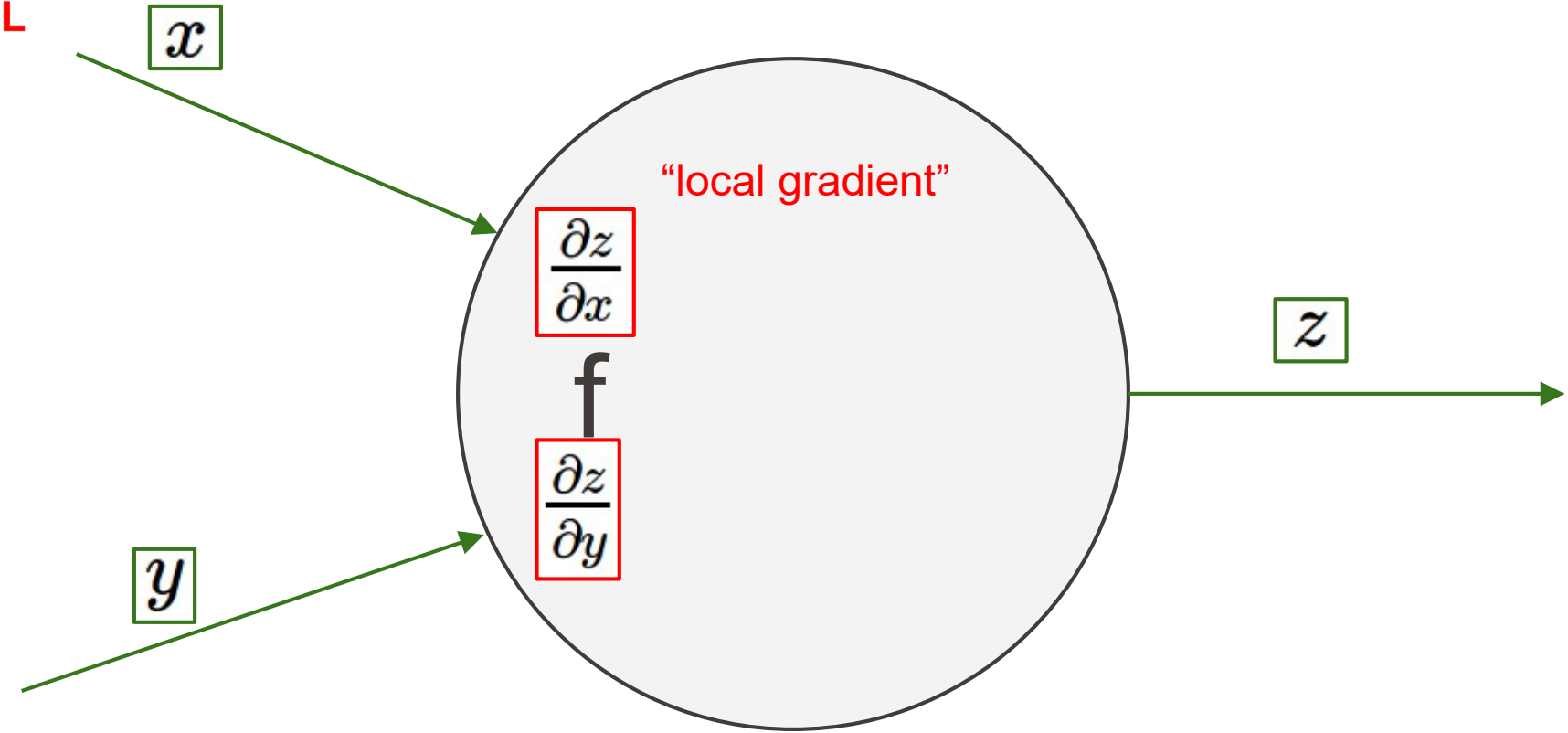
Want: $\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z}$

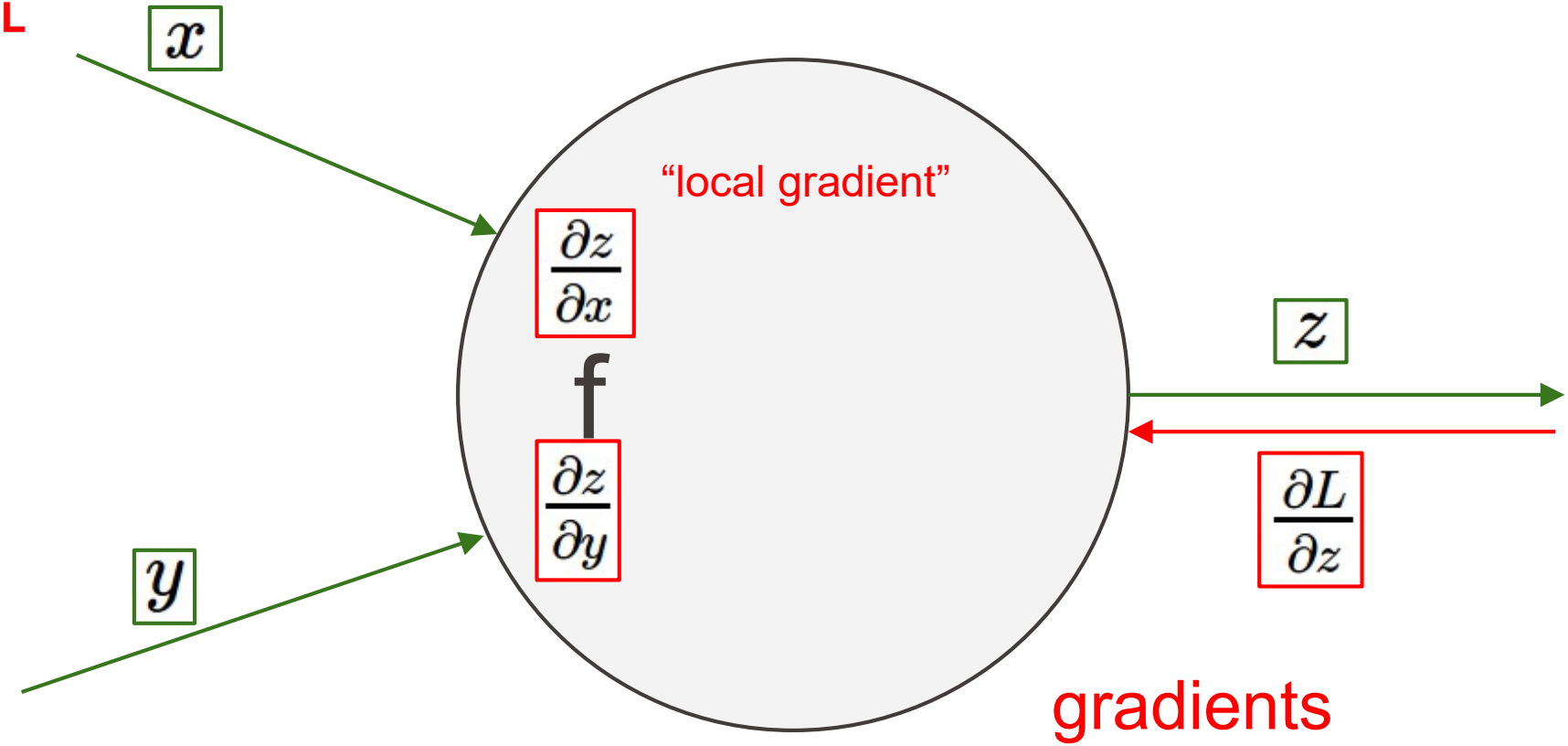


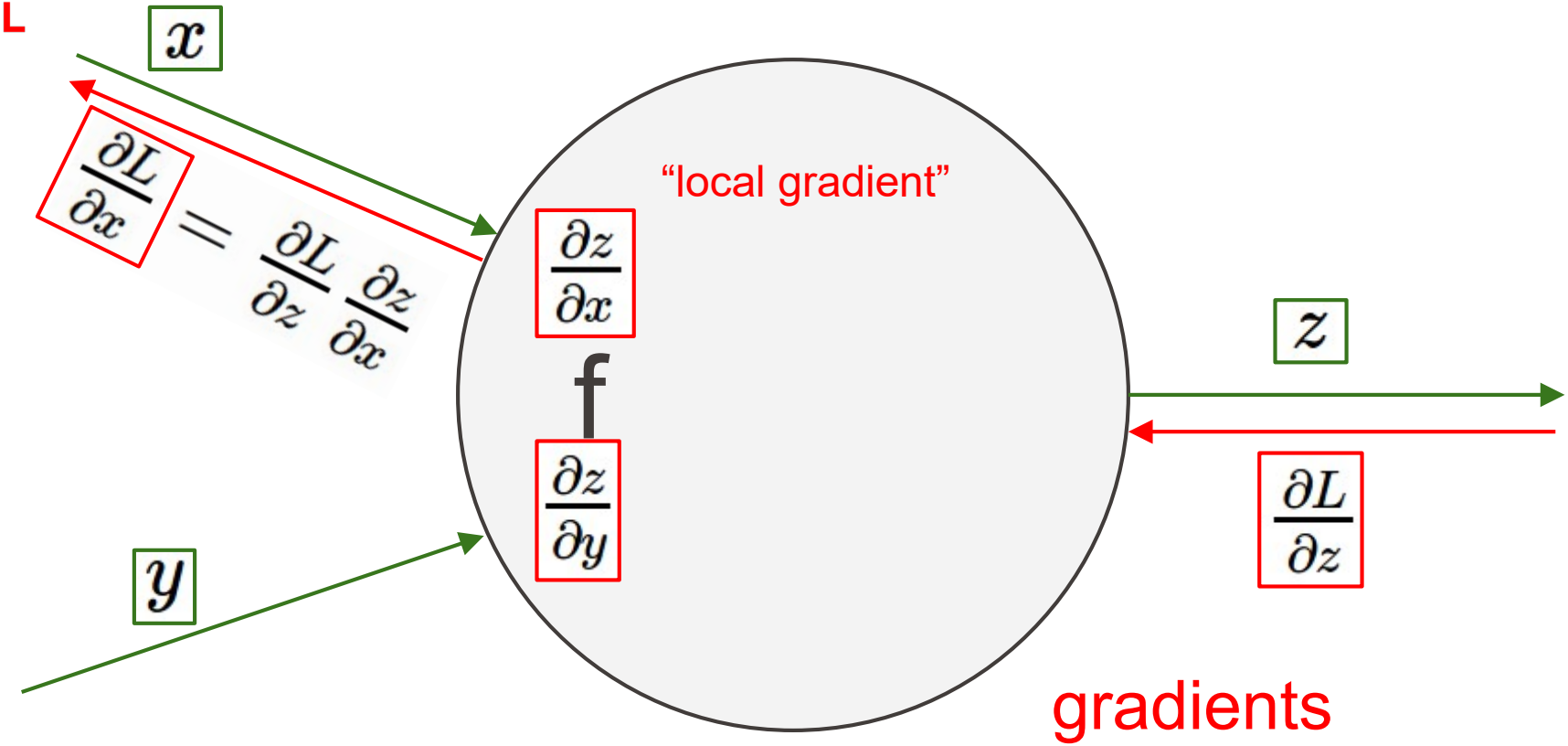
Chain rule:

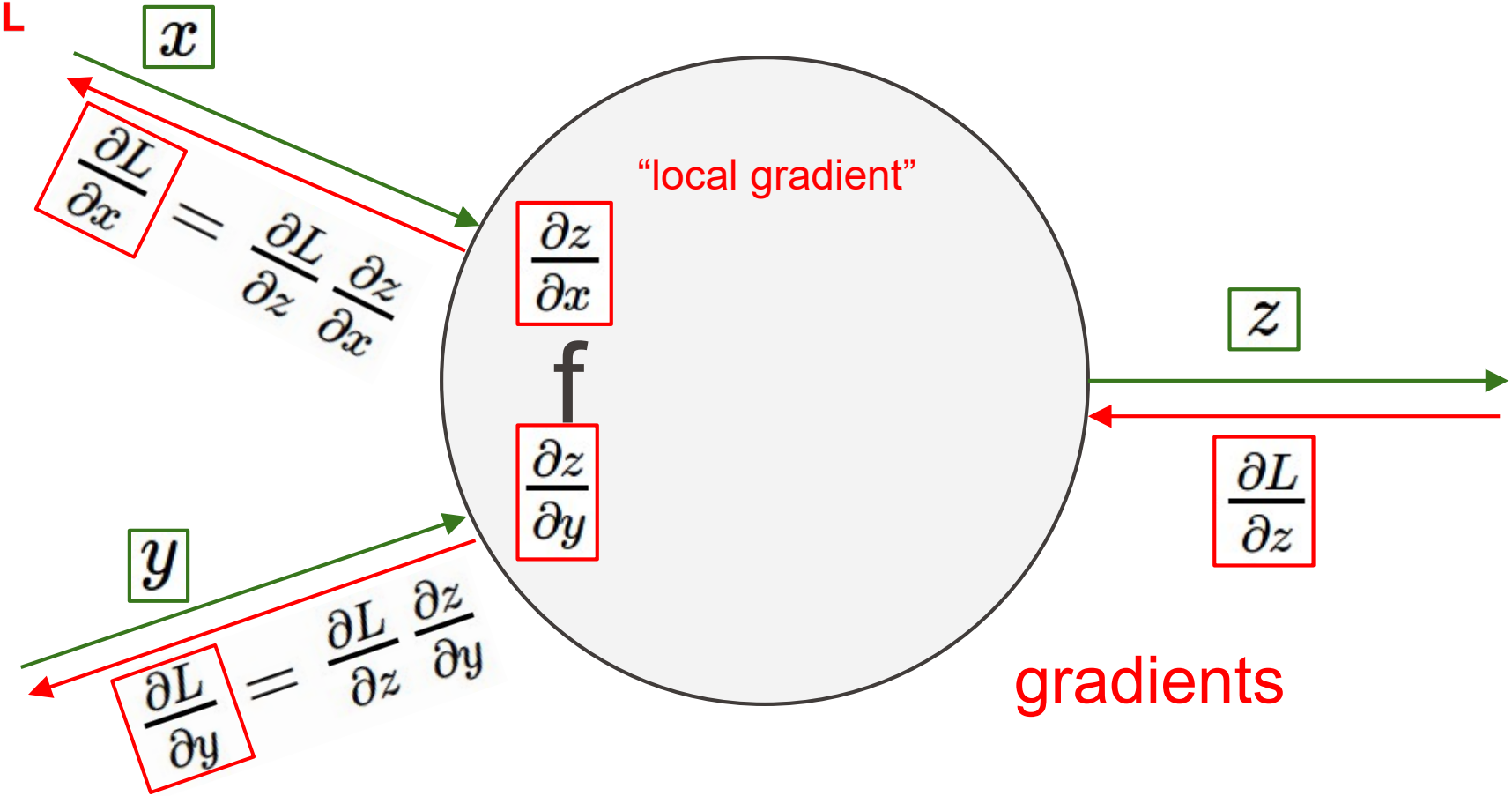
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x}$$

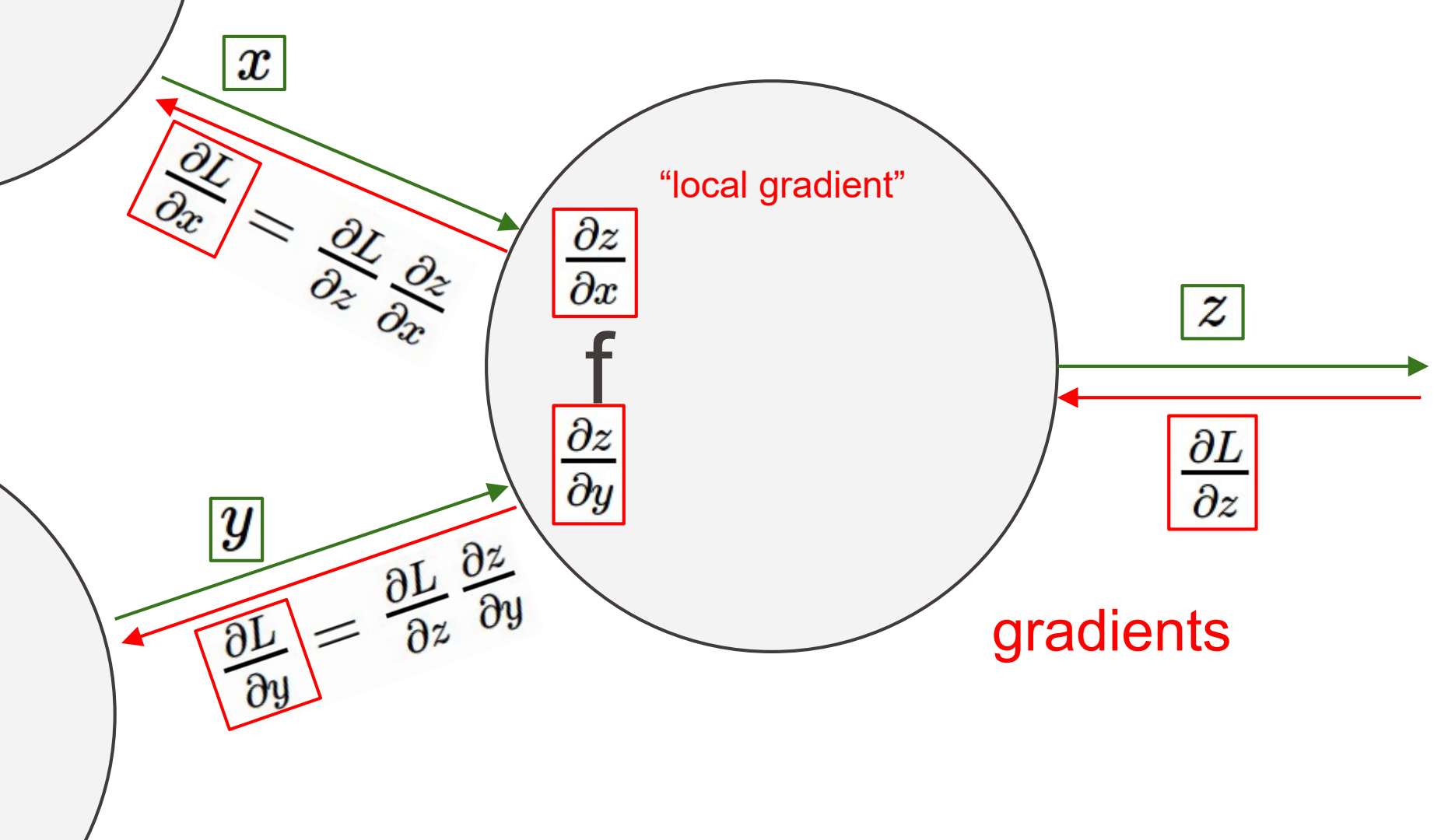

x
y
f
z


x
y
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z


x
y
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
gradients

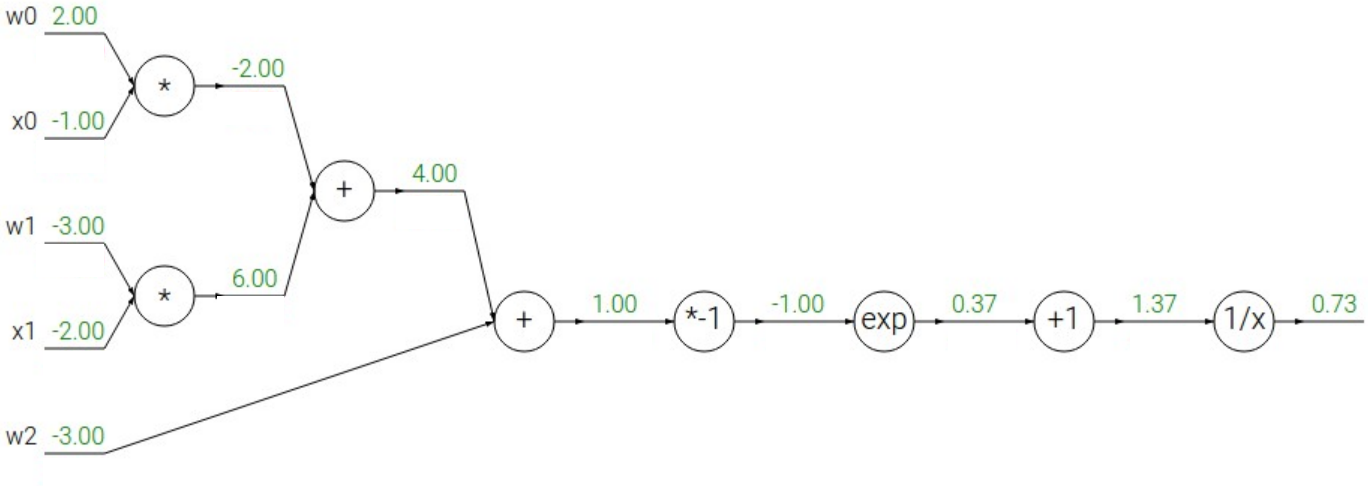

x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
y
z
$\frac{\partial L}{\partial z}$
gradients


x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients


x
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial x}$
"local gradient"
$\frac{\partial z}{\partial x}$
f
$\frac{\partial z}{\partial y}$
z
$\frac{\partial L}{\partial z}$
y
$\frac{\partial L}{\partial y} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}$
gradients

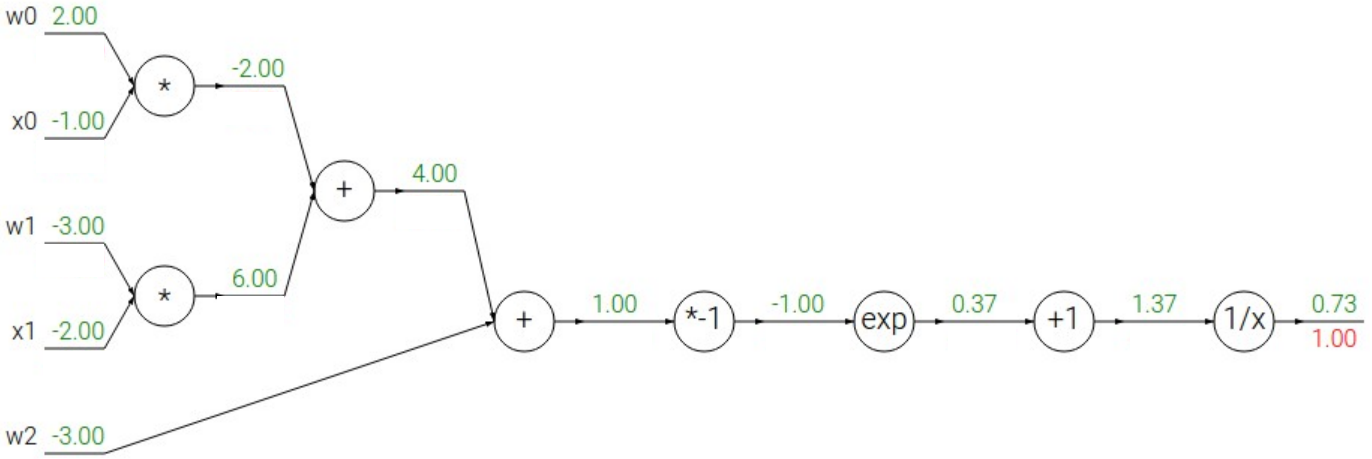
Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



Backprop

$$f(w,x) = \frac{1}{1 + e^{-(w_0 x_0 + w_1 x_1 + w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

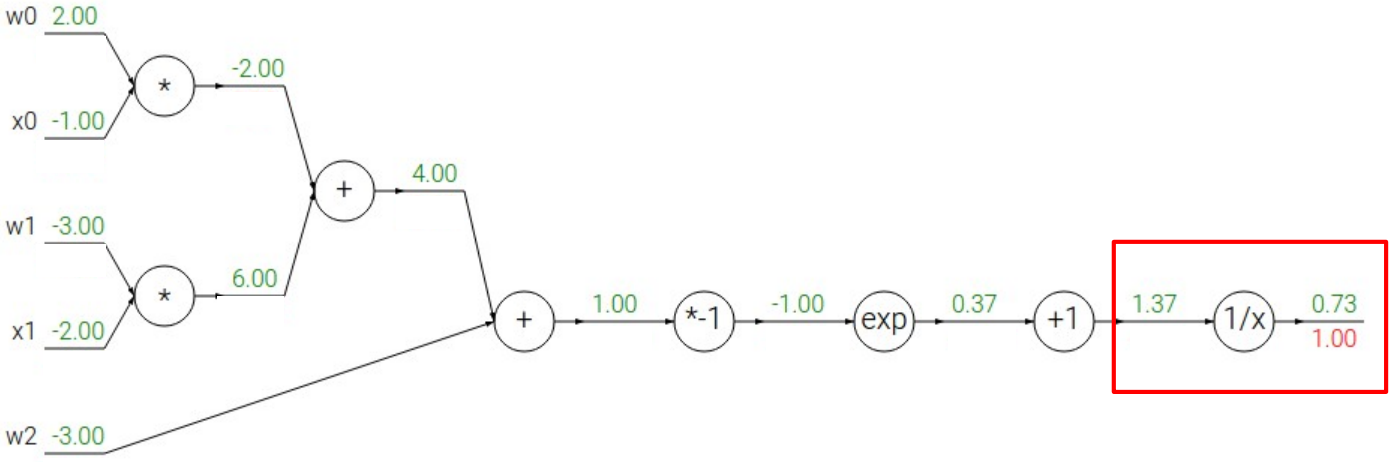
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

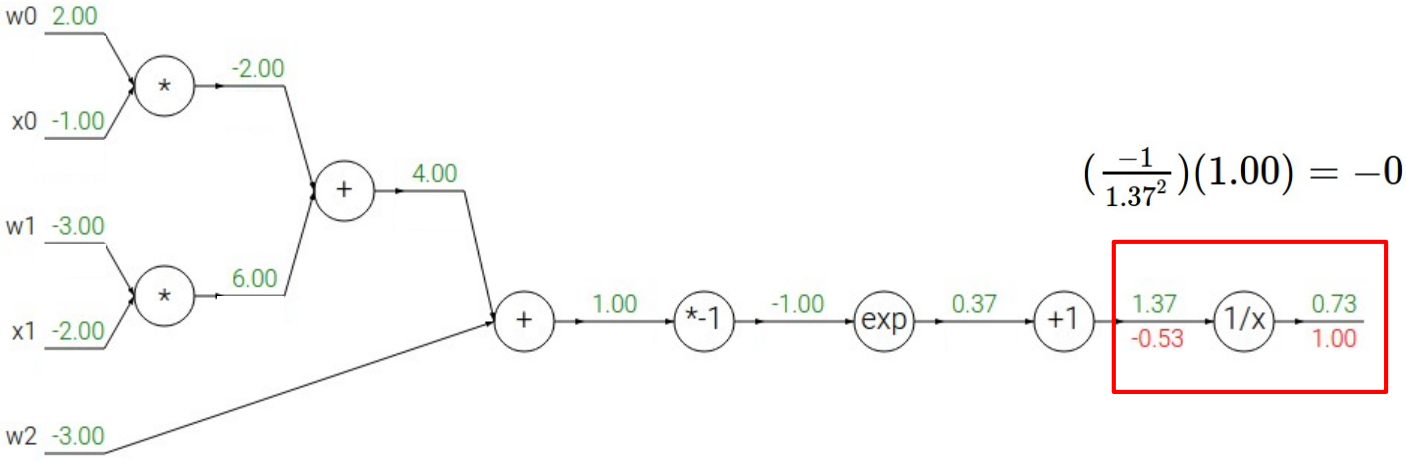
$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$(\frac{-1}{1.37^2})(1.00) = -0.53$$

$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$

$f_a(x) = ax \rightarrow \frac{df}{dx} = a$

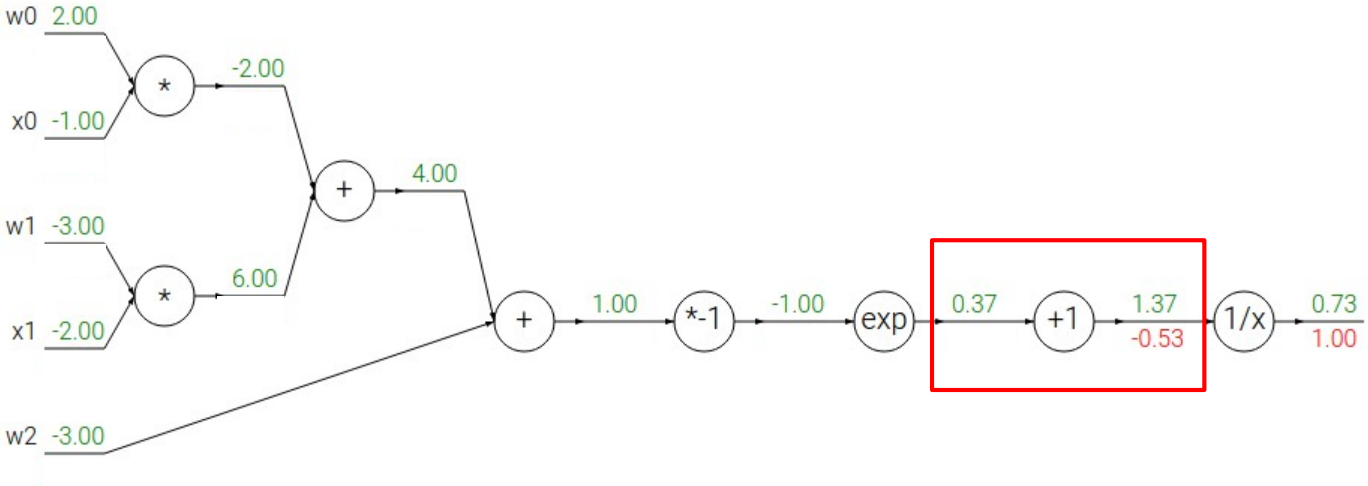
$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$

$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

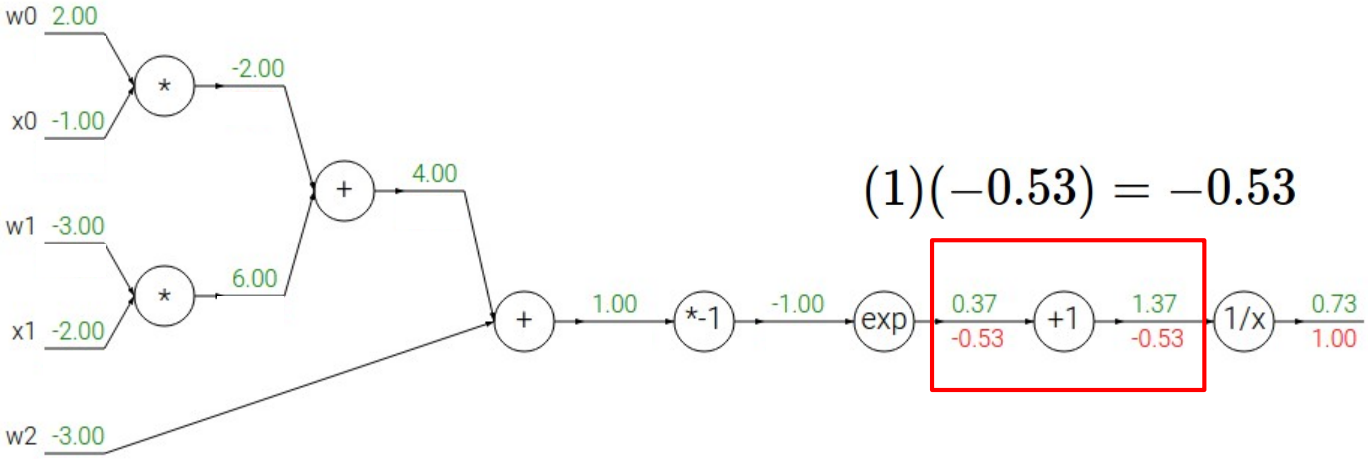
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$(1)(-0.53) = -0.53$$

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

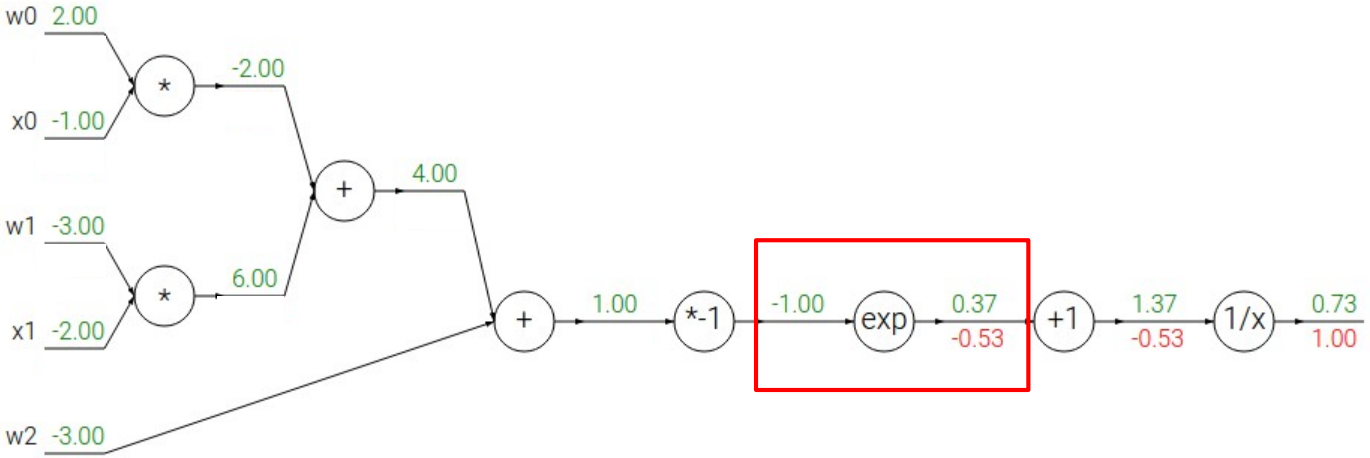
$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

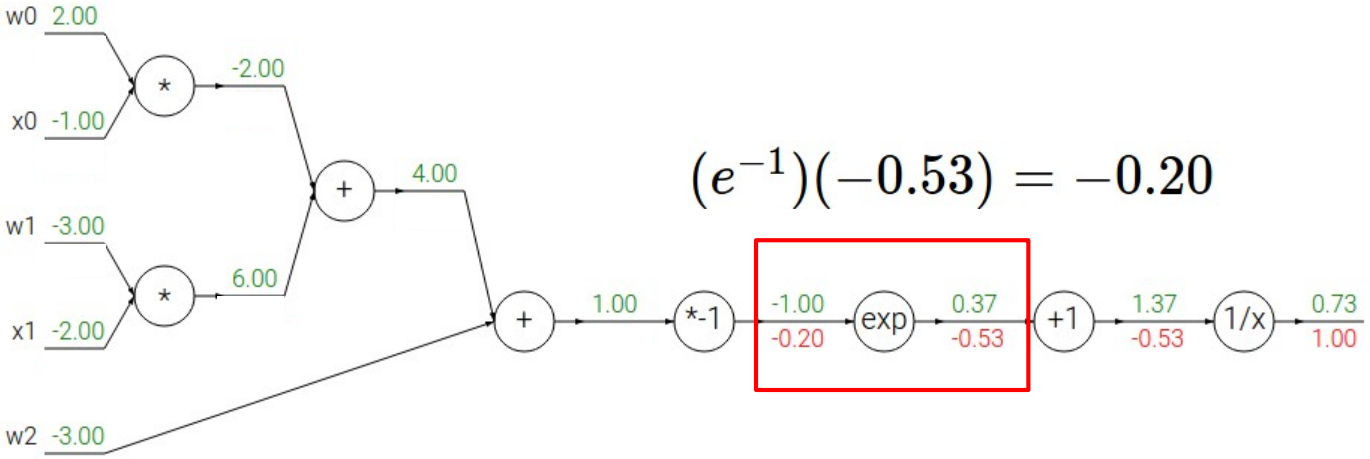
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$(e^{-1})(-0.53) = -0.20$$

$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

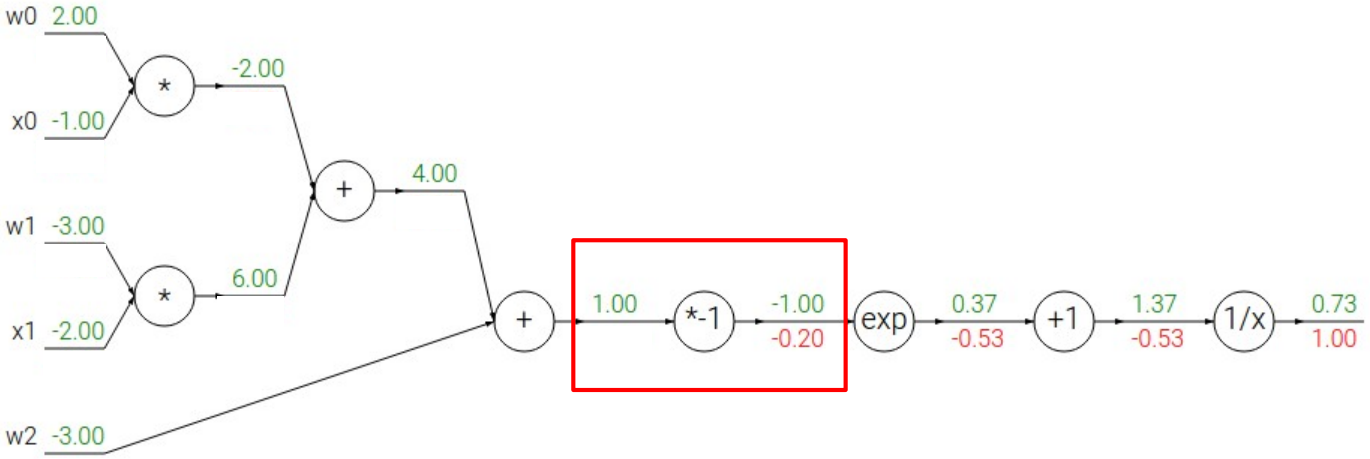
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

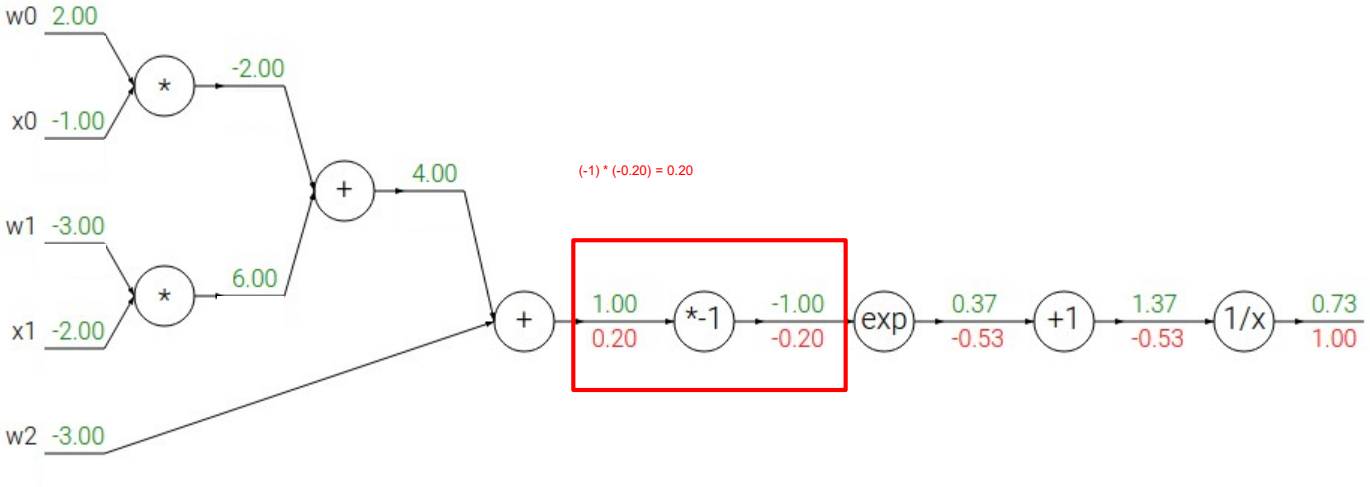
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

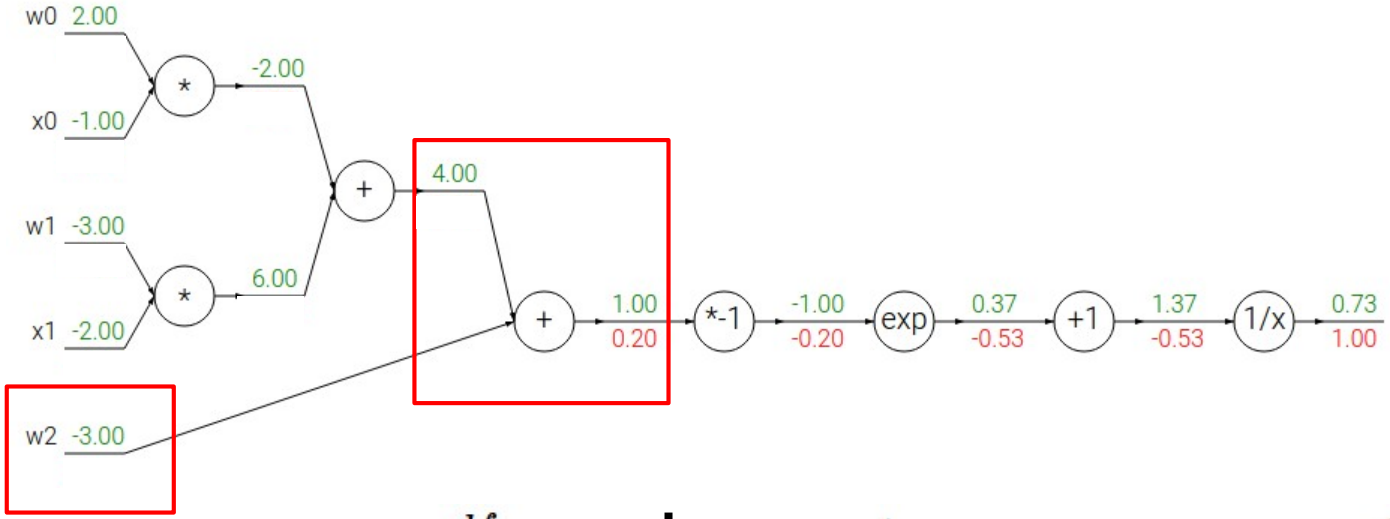
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

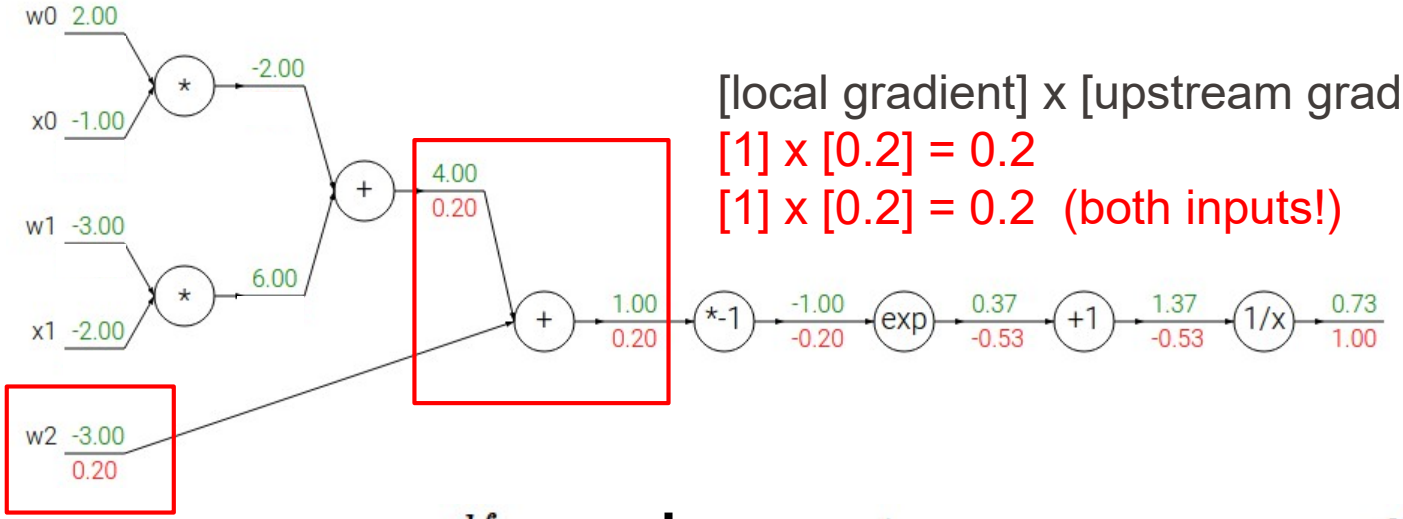
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



[local gradient] x [upstream gradient]

[1] x [0.2] = 0.2

[1] x [0.2] = 0.2 (both inputs!)

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

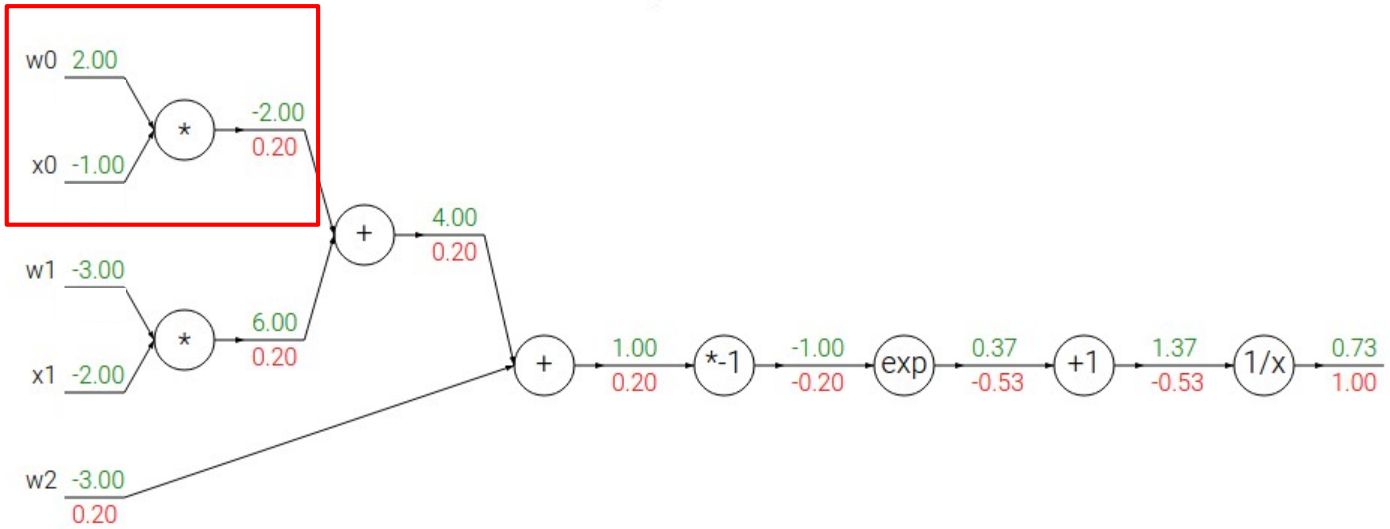
$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

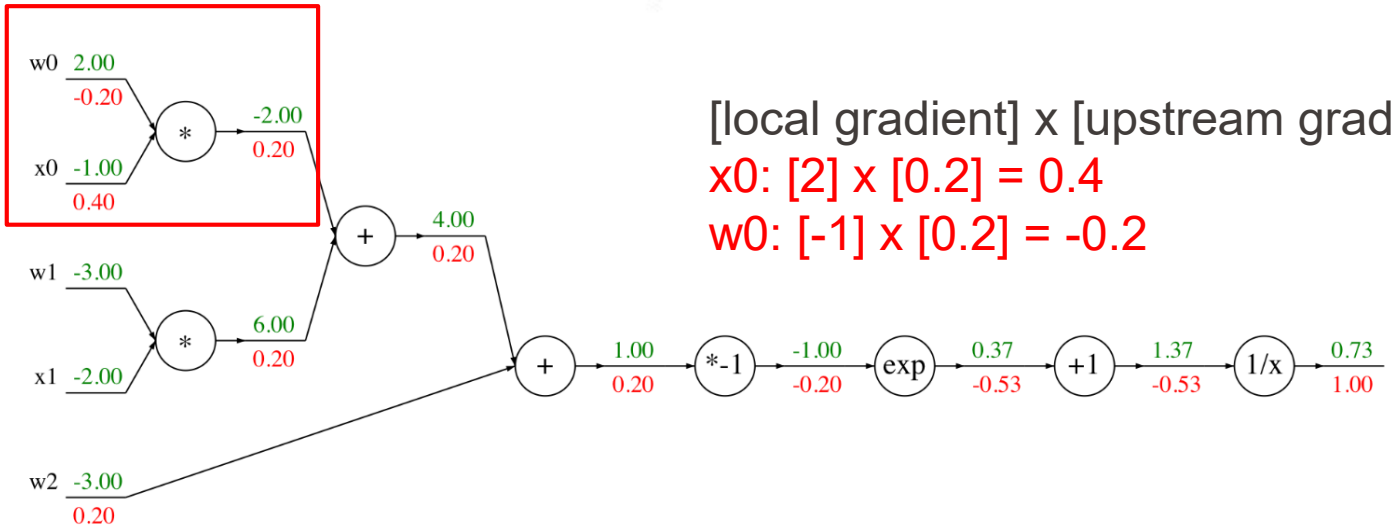
$$f(x) = \frac{1}{x} \quad \rightarrow \quad \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \quad \rightarrow \quad \frac{df}{dx} = 1$$

O. Fink / A. Alahi

Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$



[local gradient] x [upstream gradient]

x0: [2] x [0.2] = 0.4

w0: [-1] x [0.2] = -0.2

$$f(x) = e^x \rightarrow \frac{df}{dx} = e^x$$

$$f_a(x) = ax \rightarrow \frac{df}{dx} = a$$

$$f(x) = \frac{1}{x} \rightarrow \frac{df}{dx} = -1/x^2$$

$$f_c(x) = c + x \rightarrow \frac{df}{dx} = 1$$

O. Fink / A. Alahi

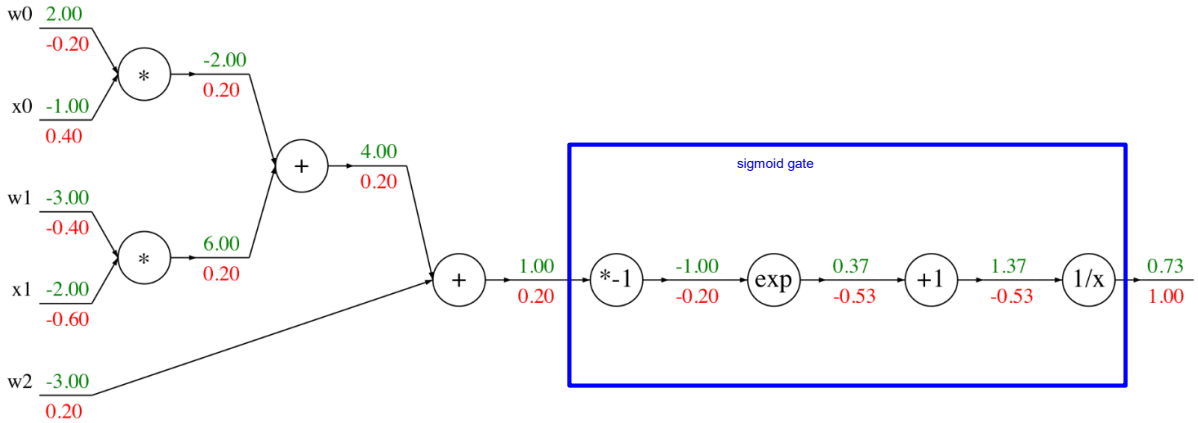
Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

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

sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



O. Fink / A. Alahi

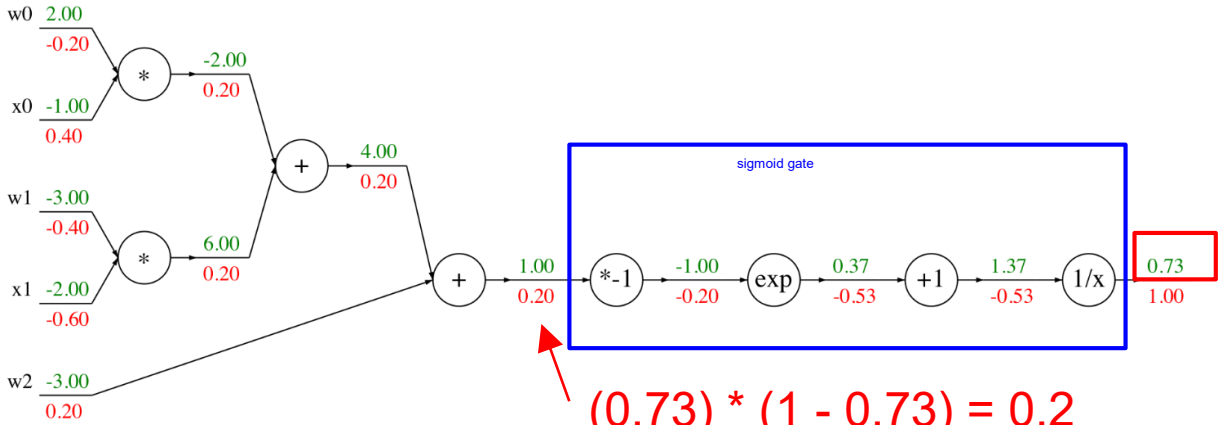
Backprop

$$f(w,x) = \frac{1}{1+e^{-(w_0x_0+w_1x_1+w_2)}}$$

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

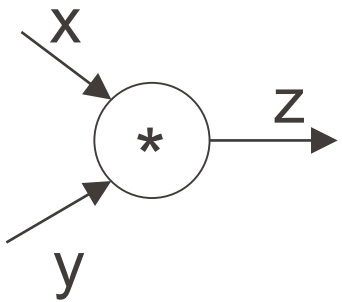
sigmoid function

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$



O. Fink / A. Alahi

Modularized implementation: forward / backward API



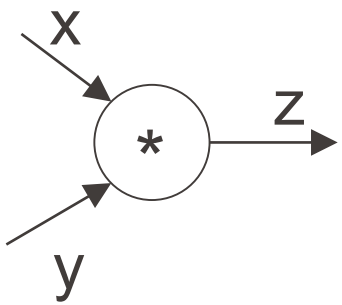
(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

$\frac{\partial L}{\partial z}$

$\frac{\partial L}{\partial x}$

Modularized implementation: forward / backward API



(x,y,z are scalars)

```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Deep learning frameworks

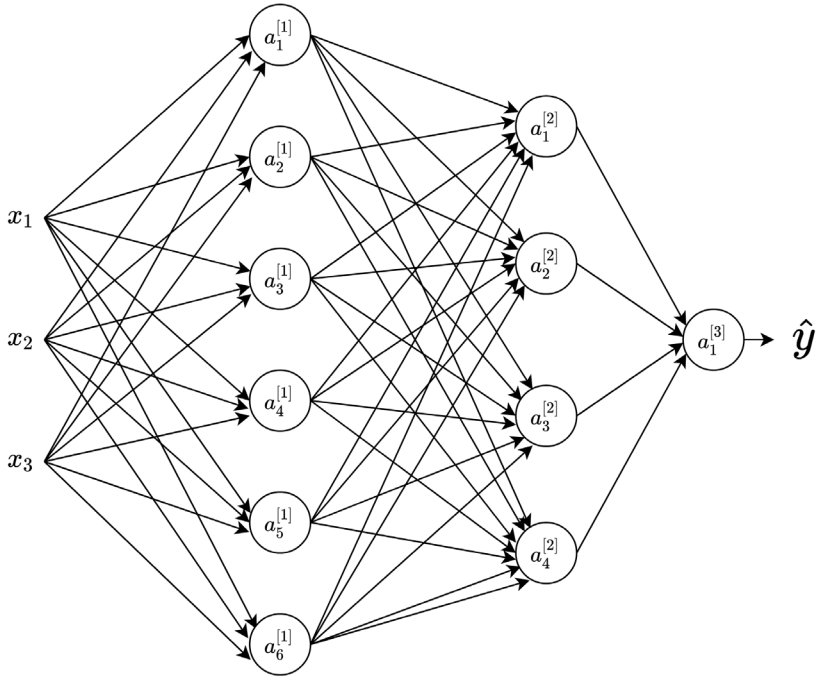
Overview

- Deep learning frameworks are used to efficiently define and train neural networks
  - Support for many types of layers, activations, loss functions, optimizers, …
  - Backpropagation computed automatically (e.g. loss.backward() in PyTorch)
  - GPU support for faster training

- Most popular frameworks today:
  - PyTorch (https://pytorch.org)
  - TensorFlow (https://www.tensorflow.org/)

Deep learning frameworks

Implementing a simple neural network in PyTorch




```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3, 6)
        self.fc2 = nn.Linear(6, 4)
        self.fc3 = nn.Linear(4, 1)

    def forward(self, x):
        # First layer
        x = self.fc1(x)
        x = F.relu(x)
        # Second layer
        x = self.fc2(x)
        x = F.relu(x)
        # Output layer
        x = self.fc3(x)
        return x
```

Recap on training a neural network

- Loop:

1. Sample a **batch of data**
2. Forward pass to **get the loss**
3. Backward pass to **calculate gradient**
4. Update parameters **using the gradient**

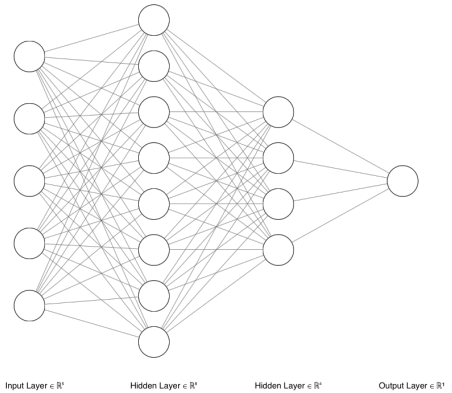
- **Forward pass** computes result of an operation and save any intermediates needed for gradient computation in memory
- **Backward pass** applies the chain rule to compute the gradient of the loss function with respect to the inputs
- **Backpropagation** = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates

Recipe for training neural networks

How to have an “efficient Gradient”?

Vanishing or exploding gradients

Definitions



1. Feed forward pass

2. Compute Loss (estimate error)
3. Backward pass

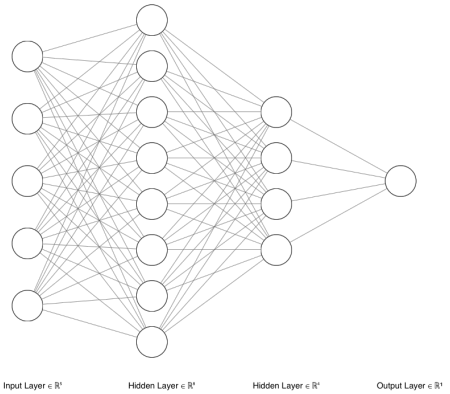
To calculate gradient
& update weight
(with stochastic gradient)

Vanishing or exploding gradients

Definitions

- **Vanishing gradient:** The update of the weights close to the input layer will become very slow, resulting in the hidden layer weights close to the input layer almost unchanged, throwing the weights close to the initialization.

- **Exploding gradient:** When the initial weight value is too large, the weight value near the input layer changes faster than the weight value near the output layer, which will cause the problem of gradient explosion.



1. Feed forward pass

2. Compute Loss (estimate error)

3. Backward pass

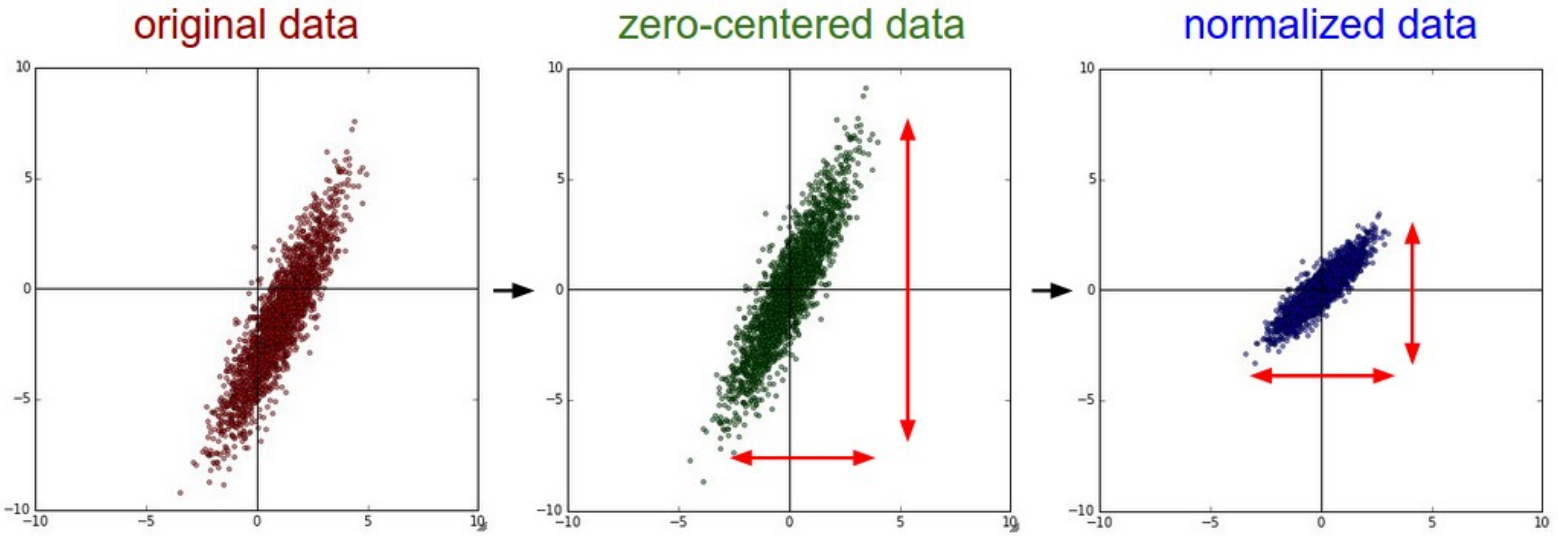
To calculate gradient
& update weight
(with stochastic gradient)

NN TIPS & TRICKS

NN TIPS & TRICKS

- Preprocess data

Data pre-processing




```
X -= np.mean(X, axis = 0)
```

```
X /= np.std(X, axis = 0)
```


O. Fink / A. Alahi

NN TIPS & TRICKS

- Preprocess data
  - **Normalize each dimension of the input**

NN TIPS & TRICKS

- Preprocess data
- Design an architecture
  - Loss function
  - Layers type
  - #layers, #Filters

› **Normalize each dimension of the input**

› **Start from state-of-the-art designs**

› **Select task appropriate loss function (pytorch losses)**

› **“Conv” for signals such as images, and more recently Transformers**

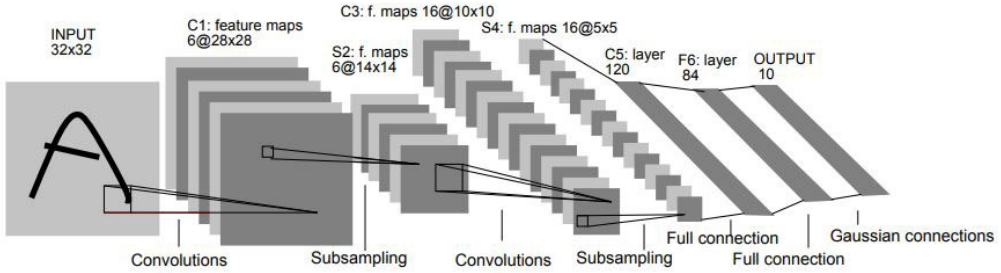
› **Start with popular designs**

Convolutional Neural Networks

Popular architectures

LeNet-5

LeCun et al. ,1998

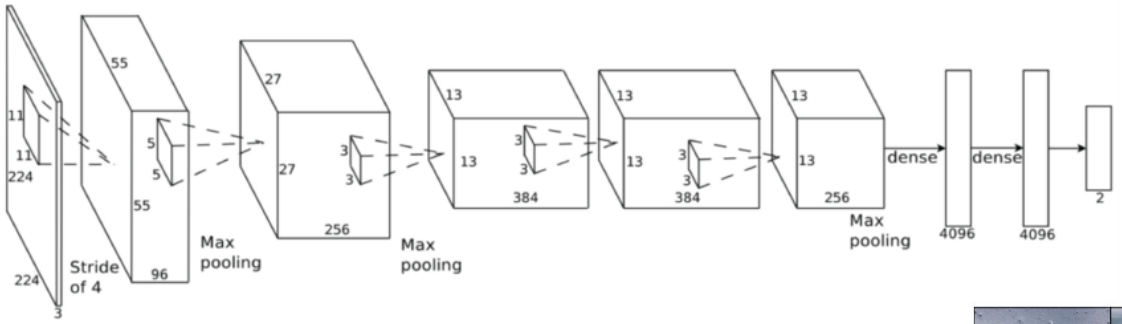



```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 6, 28, 28]             156
              ReLU-2          [-1, 6, 28, 28]               0
         MaxPool2d-3          [-1, 6, 14, 14]               0
            Conv2d-4         [-1, 16, 10, 10]           2,416
              ReLU-5         [-1, 16, 10, 10]               0
         MaxPool2d-6           [-1, 16, 5, 5]               0
            Linear-7                 [-1, 120]          48,120
              ReLU-8                 [-1, 120]               0
            Linear-9                  [-1, 84]          10,164
             ReLU-10                  [-1, 84]               0
           Linear-11                  [-1, 10]             850
          Softmax-12                  [-1, 10]               0
================================================================
Total params: 61,706
```

-1 in output shape represents the mini-batch dimension

Convolutional Neural Networks

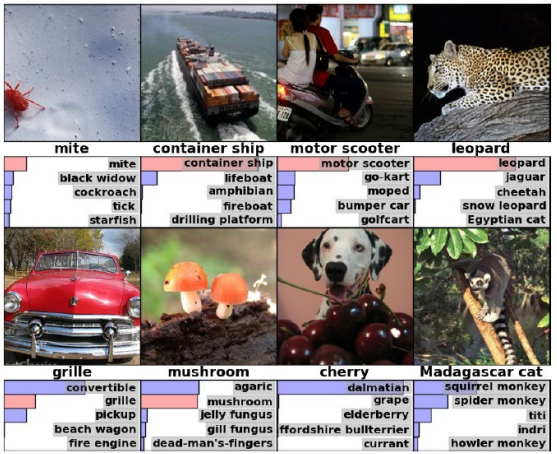
Popular architectures



AlexNet

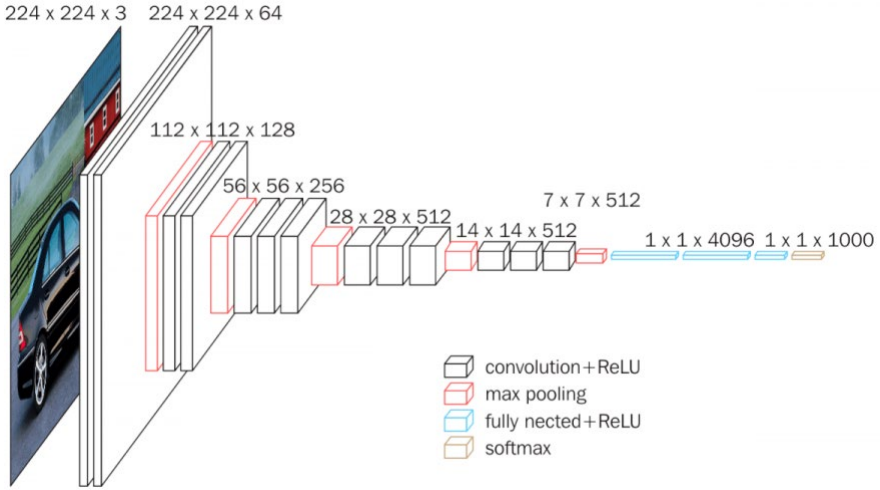
Krizhevsky et al., 2012

Winner of ImageNet Competition 2012



Convolutional Neural Networks

Popular architectures



VGG16

Simonian & Zisserman, 2014

Layer (type)	Output Shape	Param #
Conv2d-1	[-1, 64, 224, 224]	1,792
ReLU-2	[-1, 64, 224, 224]	0
Conv2d-3	[-1, 64, 224, 224]	36,928
ReLU-4	[-1, 64, 224, 224]	0
MaxPool2d-5	[-1, 64, 112, 112]	0
Conv2d-6	[-1, 128, 112, 112]	73,856
ReLU-7	[-1, 128, 112, 112]	0
Conv2d-8	[-1, 128, 112, 112]	147,584
ReLU-9	[-1, 128, 112, 112]	0
MaxPool2d-10	[-1, 128, 56, 56]	0
Conv2d-11	[-1, 256, 56, 56]	295,168
ReLU-12	[-1, 256, 56, 56]	0
Conv2d-13	[-1, 256, 56, 56]	590,080
ReLU-14	[-1, 256, 56, 56]	0
Conv2d-15	[-1, 256, 56, 56]	590,080
ReLU-16	[-1, 256, 56, 56]	0
MaxPool2d-17	[-1, 256, 28, 28]	0
Conv2d-18	[-1, 512, 28, 28]	1,180,160
ReLU-19	[-1, 512, 28, 28]	0
Conv2d-20	[-1, 512, 28, 28]	2,359,808
ReLU-21	[-1, 512, 28, 28]	0
Conv2d-22	[-1, 512, 28, 28]	2,359,808
ReLU-23	[-1, 512, 28, 28]	0
MaxPool2d-24	[-1, 512, 14, 14]	0
Conv2d-25	[-1, 512, 14, 14]	2,359,808
ReLU-26	[-1, 512, 14, 14]	0
Conv2d-27	[-1, 512, 14, 14]	2,359,808
ReLU-28	[-1, 512, 14, 14]	0
Conv2d-29	[-1, 512, 14, 14]	2,359,808
ReLU-30	[-1, 512, 14, 14]	0
MaxPool2d-31	[-1, 512, 7, 7]	0
Linear-32	[-1, 4096]	102,764,544
ReLU-33	[-1, 4096]	0
Dropout-34	[-1, 4096]	0
Linear-35	[-1, 4096]	16,781,312
ReLU-36	[-1, 4096]	0
Dropout-37	[-1, 4096]	0
Linear-38	[-1, 1000]	4,097,000
Softmax-39	[-1, 1000]	0

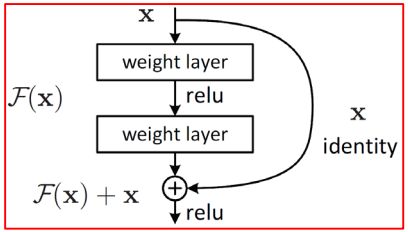
Total params: 138,357,544

Residual neural networks

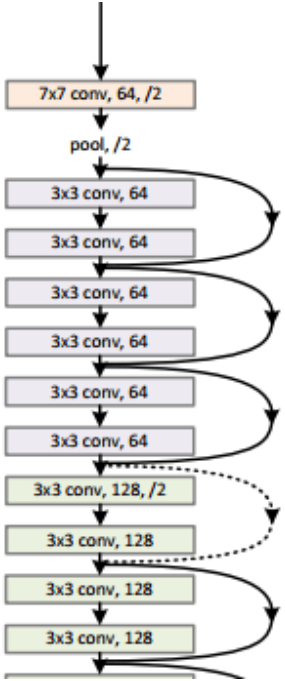
- ResNets (He et al., 2015):

Add shortcuts (skip-connections) to jump over some layers

- Deeper models are harder to optimize, and in particular, don't learn identity functions well
- Skip-connections make identity functions easier to learn, helps during training
- ResNets stack **residual blocks** on top of each other to form deep networks (e.g. ResNet-50, ResNet-101, …)



residual blocks



He and al, Deep Residual Learning for Image Recognition, 2015

Design an architecture

- https://playground.tensorflow.org/

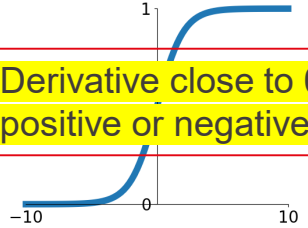
NN TIPS & TRICKS

- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **“Conv” for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**

Activation functions

Sigmoid

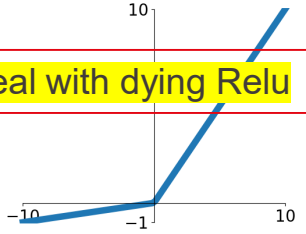
$$\sigma(x) = \frac{1}{1+e^{-x}}$$



Derivative close to 0 for large positive or negative inputs

Leaky ReLU

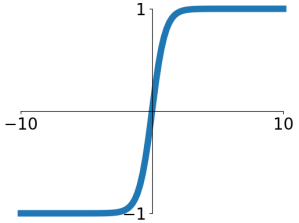
$$\max(0.1x, x)$$



Deal with dying Relu

tanh

$$\tanh(x)$$

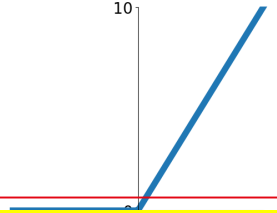


Maxout

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

ReLU

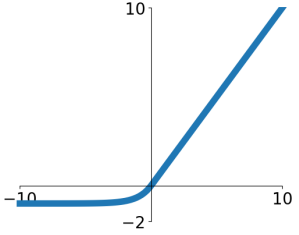
$$\max(0, x)$$



Derivative of output with respect to the input is 1 for inputs great than 0
=> Training more stable and efficient

ELU

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



NN TIPS & TRICKS

- Preprocess data — **Normalize each dimension of the input**
- Design an architecture — **Start from state-of-the-art designs**
  - Loss function — **Select task appropriate loss function (pytorch losses)**
  - Layers type — **“Conv” for signals such as images, and more recently Transformers**
  - #layers, #Filters — **Start with popular designs**
  - Activation function — **ReLu, elu**
- Sanity check
  - Gradient check
  - Loss dynamics
  - Overfit subset of the data

NN TIPS & TRICKS

- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**
- Sanity check
  - **Only once**
  - Gradient check
    - **Analytical VS num grad (while turning all regularization off)** ***(fix rand seed)***
  - Loss dynamics
    - **At chance or when regularization is increased**
  - Overfit subset of the data

Double check that the loss is reasonable:

```python
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```python
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 0.0)
print loss
```

disable regularization

```
2.30261216167
```

loss ~2.3.
"correct " for
10 classes

returns the loss and the
gradient for all parameters

Double check that the loss is reasonable:

```
def init_two_layer_model(input_size, hidden_size, output_size):
  # initialize a model
  model = {}
  model['W1'] = 0.0001 * np.random.randn(input_size, hidden_size)
  model['b1'] = np.zeros(hidden_size)
  model['W2'] = 0.0001 * np.random.randn(hidden_size, output_size)
  model['b2'] = np.zeros(output_size)
  return model
```

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
loss, grad = two_layer_net(X_train, model, y_train, 1e3)
print loss
```

crank up regularization

```
3.06859716482
```

loss went up, good. (sanity check)

O. Fink / A. Alahi

Lets try to train now…

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```


Tip: Make sure that you can overfit very small portion of the training data

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla ‘sgd’

Lets try to train now...

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

78

Tip: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-03
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Lets try to train now...


```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

Start with small regularization and find learning rate that makes the loss go down.

Lets try to train now...


```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing

Start with small regularization and find learning rate that makes the loss go down.

Lets try to train now...

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Start with small regularization and find learning rate that makes the loss go down.

Loss barely changing: Learning rate is probably too low

- **loss not going down:**

  learning rate too low

Lets try to train now...


```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000000e-06
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000000e-06
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000000e-06
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000000e-06
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000000e-06
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000000e-06
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000000e-06
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000000e-06
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000000e-06
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.000000e-06
finished optimization. best validation accuracy: 0.192000
```

Start with small regularization and find learning rate that makes the loss go down.

Loss barely changing: Learning rate is probably too low

- **loss not going down:**

  learning rate too low

Lets try to train now...

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
```

83

Start with small regularization and find learning rate that makes the loss go down.

Now let's try learning rate 1e6.

- **loss not going down:**

learning rate too low

Lets try to train now...

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide by zero en
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invalid value enc
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

Start with small regularization and find learning rate that makes the loss go down.

- **loss not going down:**

learning rate too low

- **loss exploding:**

learning rate too high

cost: NaN almost always means high learning rate...

PS: NaN also occurs because of:
- log (negative number),
- divide by zeros (variable going to zero)

Lets try to train now...

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```



```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.000000e-03
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.000000e-03
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.000000e-03
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.000000e-03
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-03
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-03
```

Start with small regularization and find learning rate that makes the loss go down.

3e-3 is still too high. Cost explodes….

- **loss not going down:**

learning rate too low

- **loss exploding:**

learning rate too high

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]

NN TIPS & TRICKS

- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**
- Sanity check
  - **Only once**
  - Gradient check
    - **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics
    - **At chance or when regularization is increased**
  - Overfit subset of the data
    - **Use small subset of the data (while regularization is off)**

NN TIPS & TRICKS

- Preprocess data — **Normalize each dimension of the input**
- Design an architecture — **Start from state-of-the-art designs**
  - Loss function — **Select task appropriate loss function (pytorch losses)**
  - Layers type — **“Conv” for signals such as images, and more recently Transformers**
  - #layers, #Filters — **Start with popular designs**
  - Activation function — **ReLu, elu**
- Sanity check — **Only once**
  - Gradient check — **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics — **At chance or when regularization is increased**
  - Overfit subset of the data — **Use small subset of the data (while regularization is off)**
- Initialize weight “smartly”

Weight initialization

Overview

How weights are initialized has an important impact on training

Q: What happens if we initialize all weights W to 0?

A: Output of each neuron of a hidden layer is identical

=> Gradient for each neuron is identical

=> Weight update for each neuron is identical

=> All neurons of a hidden layer will be identical, no better than a linear model

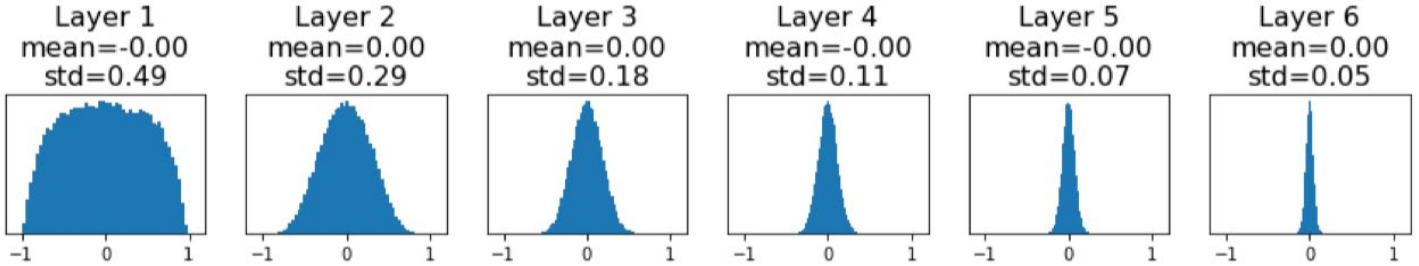
Avoid zero initialization!

Weight initialization

Initialization values

Next idea: Random initialization

- Initialize with **small** random numbers (e.g. sample from normal distribution)
  - Okay for shallow networks, problematic for deeper networks
    - Activations tend to zero for deeper network layers → small gradients, no learning



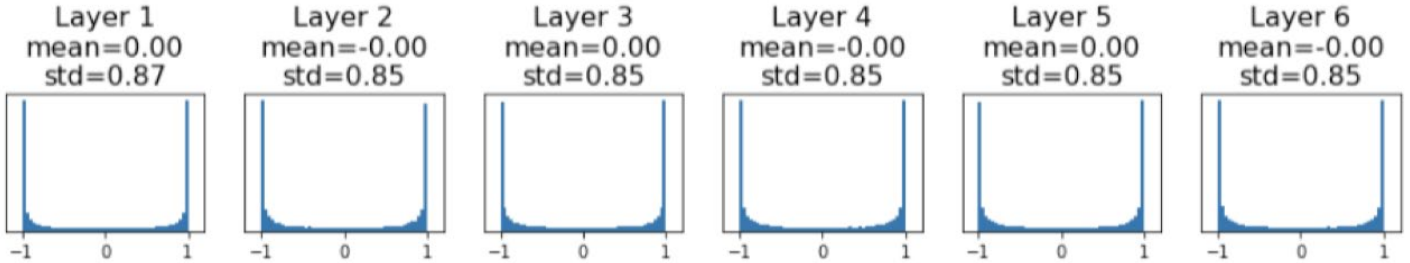
J. Johnson, Deep Learning for Computer Vision (University of Michigan) - Lecture 10

Weight initialization

Initialization values

Next idea: Random initialization

- Initialize with **larger** random numbers
  - Activations saturate → bad



J. Johnson, Deep Learning for Computer Vision (University of Michigan) - Lecture 10

=> Need to find initialization values that are "just right"

Weight initialization

Initialization values

Xavier (Glorot) initialization [Glorot et al., 2010]:

Sample from $\mathcal{N}(0, \sigma^2)$, with $\sigma = \sqrt{\frac{2}{a+b}}$,

where a is the number of input neurons, and b is the number of output neurons

- Good initialization for network with $tanh$ activations

Kaiming (He) initialization [He et al., 2015]:

Sample from $\mathcal{N}(0, \sigma^2)$, with $\sigma = \sqrt{\frac{2}{a}}$, where a is the number of input neurons

- Good initialization for network with $ReLU$ activations
- More info: https://paperswithcode.com/method/he-initialization

Weight initialization

Resources

- **(Must read)** Great interactive lecture notes on initialization by Katanforoosh & Kunin: https://www.deeplearning.ai/ai-notes/initialization/

NN TIPS & TRICKS

- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**
- Sanity check
  - **Only once**
  - Gradient check
    - **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics
    - **At chance or when regularization is increased**
  - Overfit subset of the data
    - **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly"
  - **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**

NN TIPS & TRICKS

- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**
- Sanity check
  - **Only once**
  - Gradient check
    - **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics
    - **At chance or when regularization is increased**
  - Overfit subset of the data
    - **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly"
  - **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization

Batch Normalization

O. Fink / A. Alahi

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

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

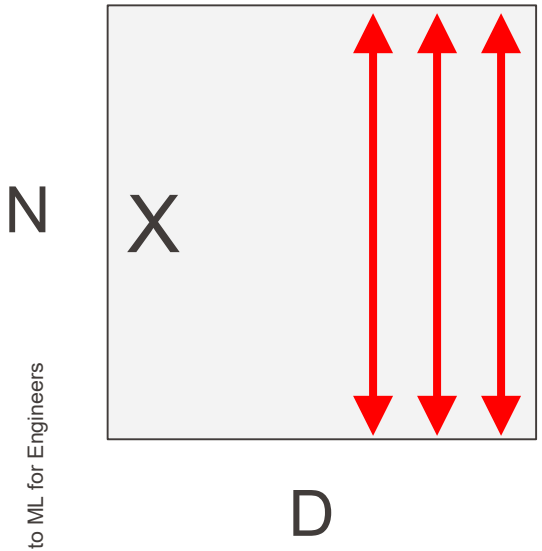
this is a vanilla differentiable function...

[Ioffe and Szegedy, 2015]

Batch Normalization

O. Fink / A. Alahi

"you want unit gaussian activations?
just make them so."



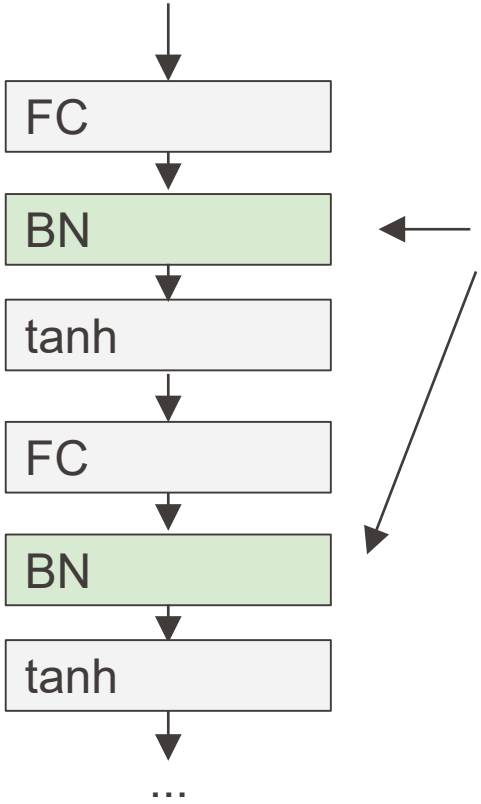
1. compute the empirical mean and variance independently for each dimension.

2. Normalize

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

[Ioffe and Szegedy, 2015]

Batch Normalization



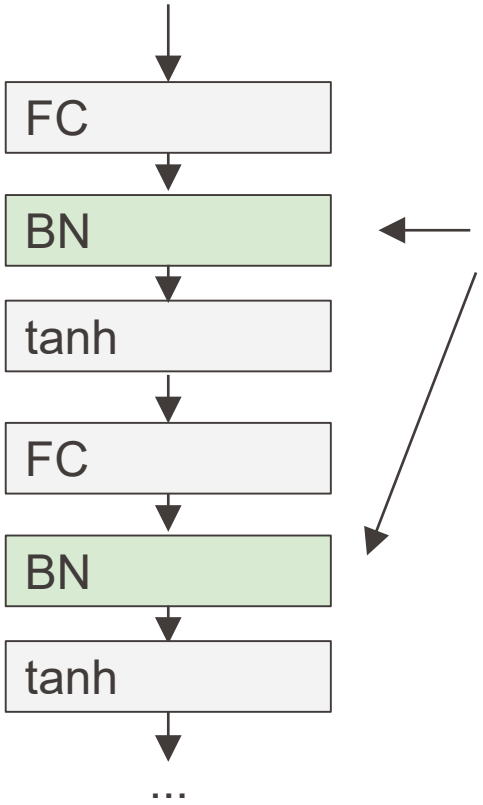
Usually inserted after Fully Connected or Convolutional layers, and before nonlinearity.

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

[Ioffe and Szegedy, 2015]

O. Fink / A. Alahi

Batch Normalization



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

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

[Ioffe and Szegedy, 2015]

O. Fink / A. Alahi

Batch Normalization

Normalize:

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

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

[Ioffe and Szegedy, 2015]

Batch Normalization

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

[Ioffe and Szegedy, 2015]

Batch Normalization

O. Fink / A. Alahi

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g., can be estimated during training with running averages)

[Ioffe and Szegedy, 2015]

Batch Normalization

- Lubana, E. S., Dick, R., & Tanaka, H. (2021). Beyond BatchNorm: Towards a unified understanding of normalization in deep learning. Neural Information Processing Systems

NN TIPS & TRICKS

- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**
- Sanity check
  - **Only once**
  - Gradient check
    - **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics
    - **At chance or when regularization is increased**
  - Overfit subset of the data
    - **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly"
  - **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization
  - **Try with and without. Test also Layer norm, group norm**

NN TIPS & TRICKS

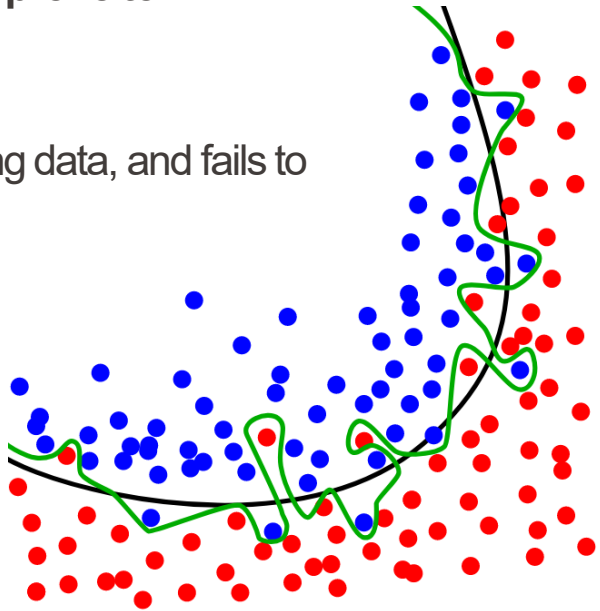
- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**
- Sanity check
  - **Only once**
  - Gradient check
    - **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics
    - **At chance or when regularization is increased**
  - Overfit subset of the data
    - **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly"
  - **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization
  - **Try with and without. Test also Layer norm, group norm**
- Regularization strategies

Regularization
Introduction

- Deep neural networks can learn very complex functions → **prone to overfitting**

- **Overfitting:** when a model learns the "noise" of the training data, and fails to generalize → **high variance, low bias**
  - Great performance on training set
  - Much worse on test set

- Many **regularization** techniques exist to limit overfitting:
  - Some quite generic (e.g. data augmentation)
  - Some exclusive to neural nets (e.g. dropout)



Green: Overfitting
Black: Regularized model

Regularization

Getting more data

- Overfitting is caused by noisy data & a complex model

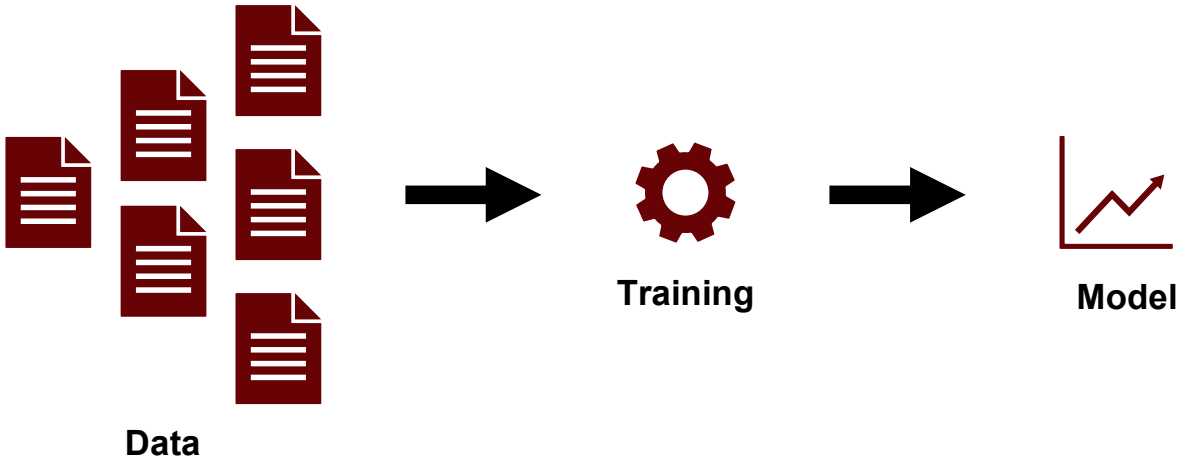
Regularization

Getting more data

- Overfitting is caused by **noisy data** & a complex model

- **Solution:** Gather more data to reduce noise

✦ The more data we collect, the better the model can identify the underlying phenomenon that generates the data



Regularization
Getting more data

- **Drawback:** getting new data can be very difficult or costly
  - Some data is finite (e.g. civil data)
  - Labeling data takes time, need to do it manually
  - Some datasets already have millions of samples, gathering a few new samples won't do much

The ImageNet dataset contains over 14 million images!

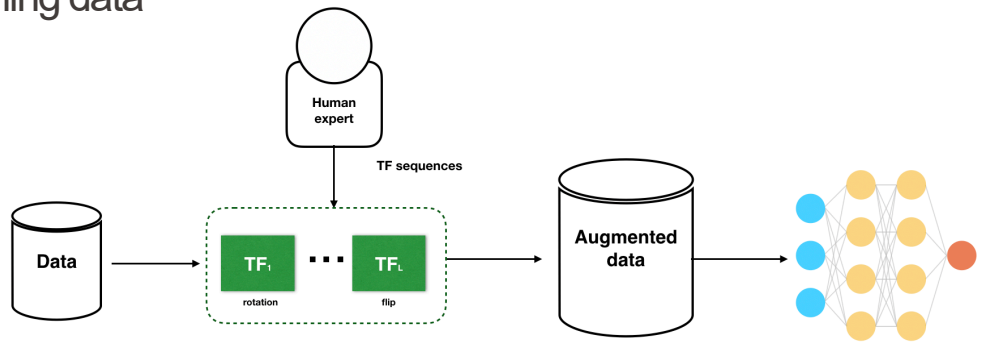
Regularization
Data augmentation

- Deep neural nets have millions / billions of parameters

→requires a proportional amount of training samples, which can be hard to obtain

- **Solution:** Use data augmentation!

- **Data augmentation:** Artificially generate new training samples by slightly modifying existing training data



S. Li, Automating Data Augmentation: Practice, Theory and New Direction, 2020

Regularization
Data augmentation

- Data augmentation:
- Artificially generate new training samples by slightly modifying existing data

- Some commonly used techniques (for images):
  - **Cropping**
  - **Flipping**
  - **Rotating**
  - **Color / contrast jittering**
  - **Adding noise**

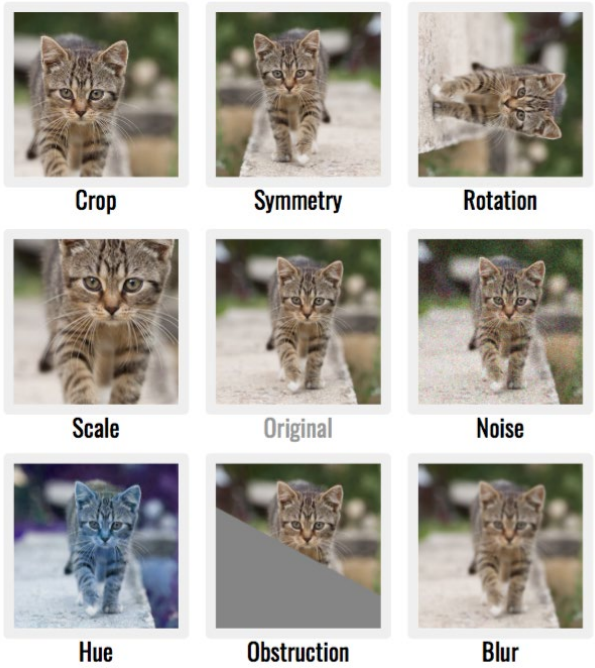
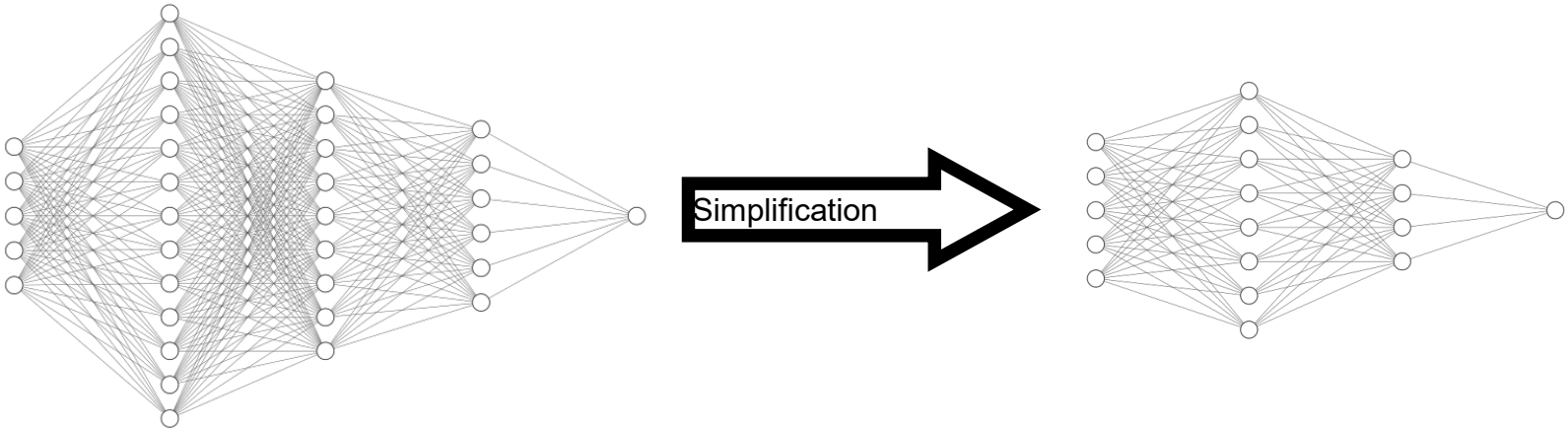


Image credit: Hackernoon

Regularization
Using less parameters

- Overfitting is caused by noisy data & **a complex model**

- **Solution:** Simplify the model by reducing the number of parameters
  - Reduce number of hidden layers (depth)
  - Reduce number of neurons in each hidden layer (width)



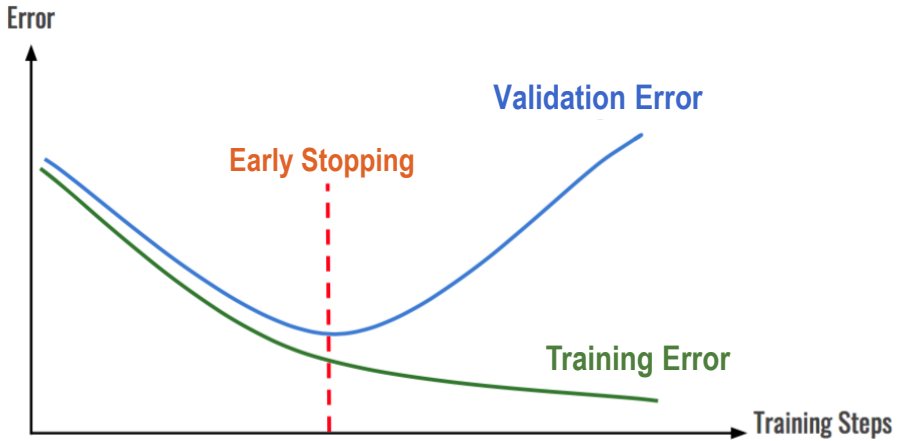
Regularization

Early stopping

- Overfitting → low training error, high validation & test error

- **Early stopping:** Stop the training process when the the validation error starts increasing

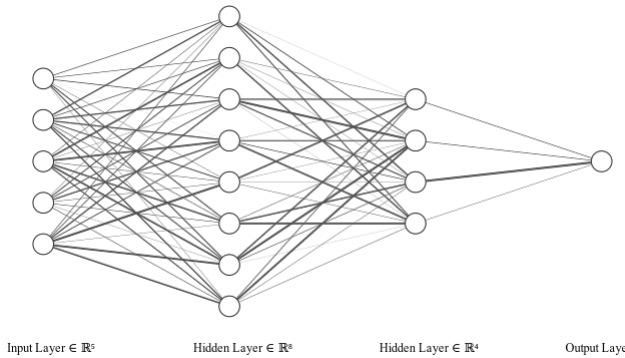
✦ Simple technique, but requires frequent evaluation on the validation set



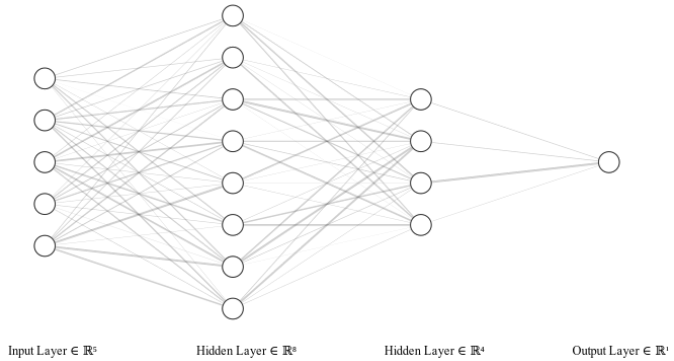
Regularization
L2, L1 & elastic net

Limit overfitting by adding a regularization term to the cost function

- Constrains the weights (high value gets penalized)
- λ: hyper-parameter which tunes the strength of the regularization
  - bigger λ = more regularization



No regularization



With L2 regularization (weight decay)

Regularization

L2

L2 regularization, also known as ridge or weight decay

- One of the most popular regularization techniques in ML
  - Used for linear regression, logistic regression, neural nets, SVMs, …

For neural nets, define: $\| \mathbf{W} \|_2^2 = \sum_{l=1}^{L} \| \mathbf{W}^{[l]} \|_2^2$ where $\| \mathbf{W}^{[l]} \|_2^2 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} (W_{i,j}^{[l]})^2$

Sum the square of each weight

$$J = \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, \hat{y}^{(i)}) + \lambda \| \mathbf{W} \|_2^2$$

Loss function: predictions should match training data

L2 Regularization term: model should be "simple"

Regularization

L1

L1 regularization, also known as lasso (Least Absolute Shrinkage Selector Operator):

- Tends to force weights to 0, which increases sparsity

For neural nets, define: $\| \mathbf{W} \|_1 = \sum_{l=1}^{L} \| \mathbf{W}^{[l]} \|_1$ where $\| \mathbf{W}^{[l]} \|_1 = \sum_{i=1}^{n^{[l-1]}} \sum_{j=1}^{n^{[l]}} |W_{i,j}^{[l]}|$

Sum the absolute value of each weight

$$J = \frac{1}{N} \sum_{i=1}^{N} L(y^{(i)}, \hat{y}^{(i)}) + \lambda \| \mathbf{W} \|_1$$

Loss function **L1 Regularization term**

Regularization
Elastic net

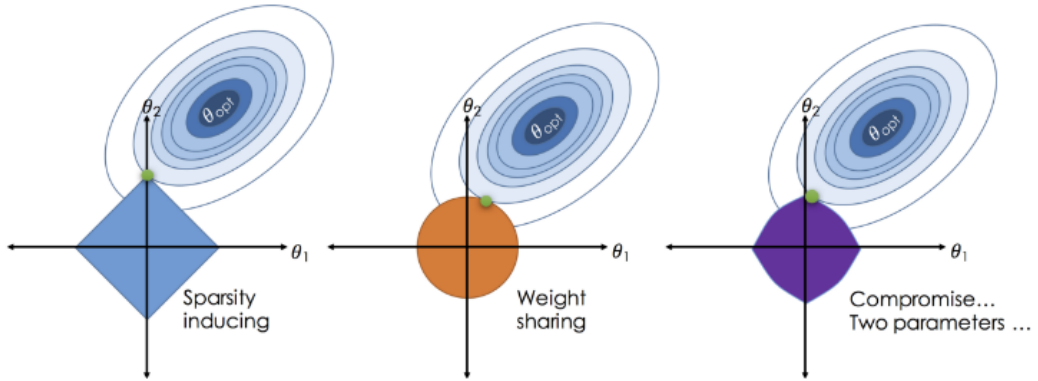
Elastic net regularizations combines L1 and L2

$$J = \frac{1}{N}\sum_{i=1}^{N} L(y^{(i)}, \hat{y}^{(i)}) + \lambda_2 \parallel \mathbf{W} \parallel_2^2 + \lambda_1 \parallel \mathbf{W} \parallel_1$$

Loss function

L2 regularization term

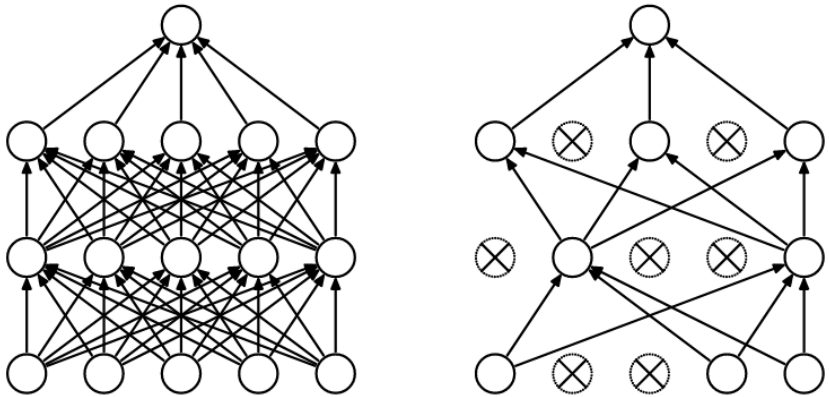
L1 regularization term



L1 L2 Elastic net (L1 + L2)

Regularization
Dropout - Overview

- Dropout:
- Randomly drop nodes (along with connections) with probability $1-p$ during training



(a) Standard Neural Net (b) After applying dropout.

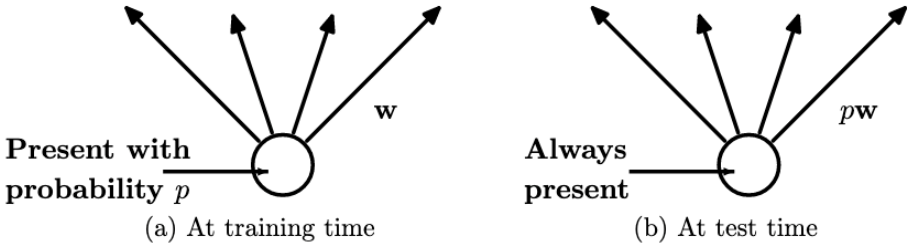
Srivastava, Hinton et al. , Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2012

Regularization
Dropout - Implementation

- At training time:
- At each iteration, keep node with probability p
  - Dropped nodes change at each iteration
  - Lower $p \rightarrow$ stronger regularization

p is a hyperparameter

- At test time:
- All nodes are present, but weights are scaled by p (i.e. w becomes pw)



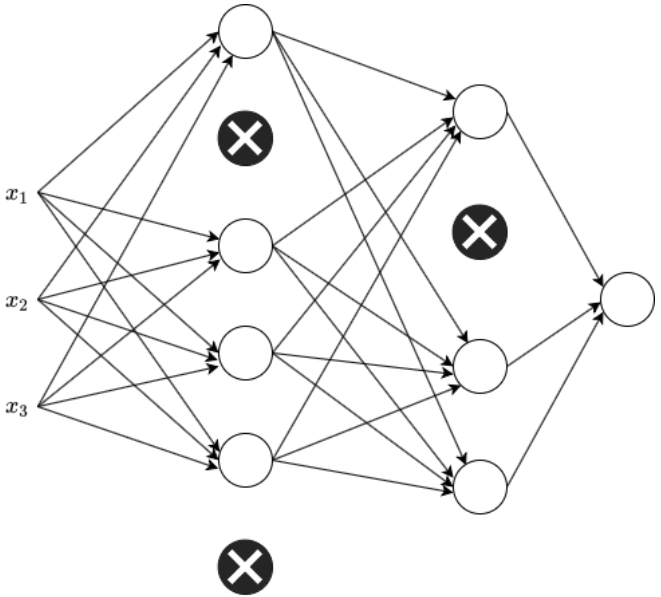
(a) At training time (b) At test time

Regularization
Dropout - Implementation

When implemented, p usually varies per layer

Example: $p^{[0]} = 1, p^{[1]} = 0.5, p^{[2]} = 0.75$

where $p^{[l]}$ is the probability of keeping a node in the l^{th} layer



Training: Iteration 1

Note:

In some literature, p instead refers to the probability of dropping the nodes.

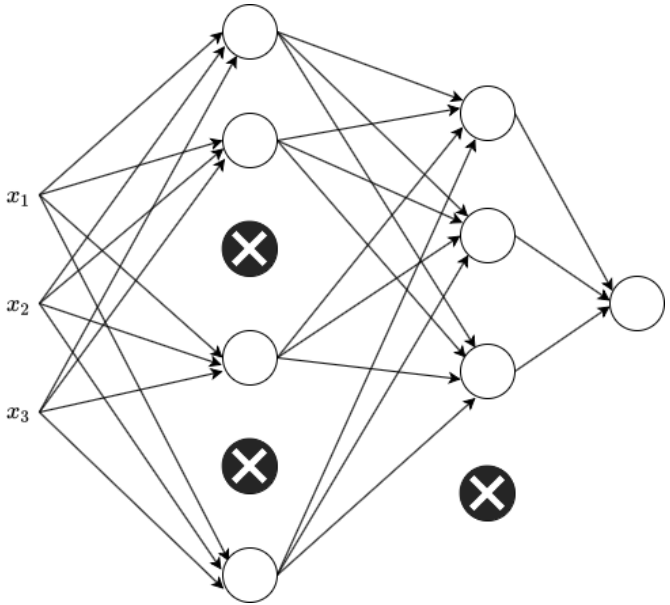
Weights are scaled by $(1 - p)$ with this notation

Regularization
Dropout - Implementation

When implemented, p usually varies per layer

Example: $p^{[0]} = 1, p^{[1]} = 0.5, p^{[2]} = 0.75$

where $p^{[l]}$ is the probability of keeping a node in the l^{th} layer



Training: Iteration 2

Note:

In some literature, p instead refers to the probability of dropping the nodes.

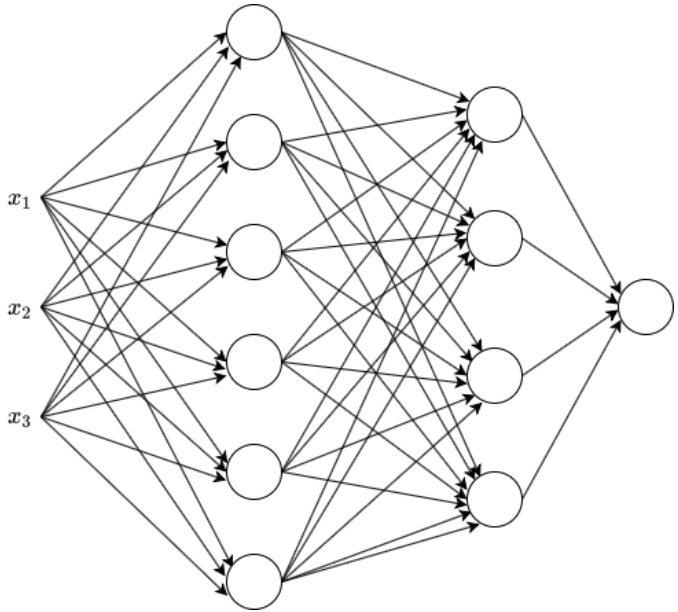
Weights are scaled by $(1 - p)$ with this notation

Regularization
Dropout - Implementation

When implemented, p usually varies per layer

Example: $p^{[0]} = 1, p^{[1]} = 0.5, p^{[2]} = 0.75$

where $p^{[l]}$ is the probability of keeping a node in the l^{th} layer



Test time

Note:

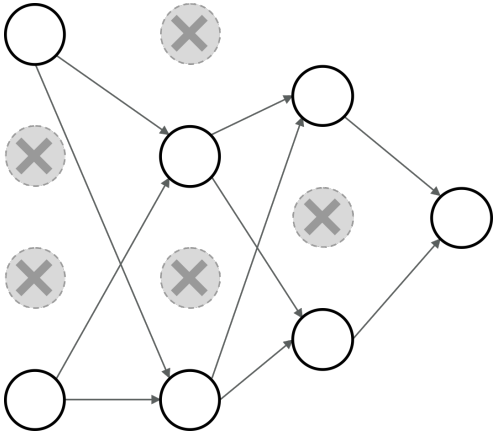
In some literature, p instead refers to the probability of dropping the nodes.

Weights are scaled by $(1 - p)$ with this notation

Regularization

Dropout - Intuition

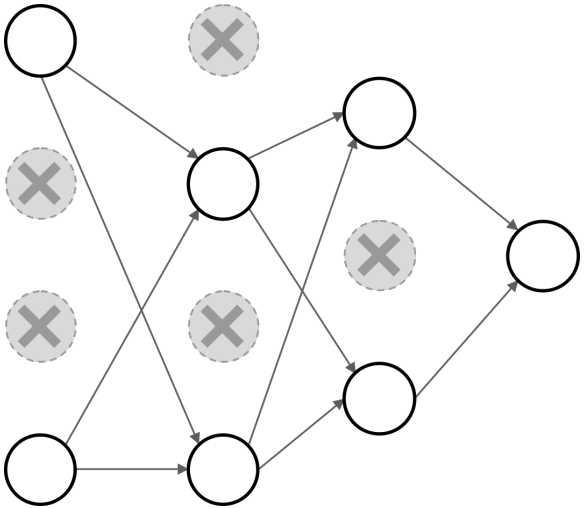
- How can dropout possibly be a good idea?
  - It forces the network to have a redundant representation
  - Prevents co-adaptation of features



The network can't rely on any specific feature as they are randomly dropped during training → needs redundancy

Regularization

Dropout - Intuition



- Another interpretation:
- Dropout is training a large ensemble of models (that share parameters)

- Each binary mask is one model

- An FC layer with 4096 units has
- $2^{4096} \sim 10^{1233}$ possible masks!
- Only $\sim 10^{82}$ atoms in the universe...

Regularization

Resources

- Data augmentation:
  http://ai.stanford.edu/blog/data-augmentation/

- **(Must read)** L1 vs L2 regularization:
  https://developers.google.com/machine-learning/crash-course/regularization-for-sparsity/l1-regularization

- Dropout paper:
  https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

NN TIPS & TRICKS

- Preprocess data — **Normalize each dimension of the input**
- Design an architecture — **Start from state-of-the-art designs**
  - Loss function — **Select task appropriate loss function (pytorch losses)**
  - Layers type — **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters — **Start with popular designs**
  - Activation function — **ReLu, elu**
- Sanity check — **Only once**
  - Gradient check — **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics — **At chance or when regularization is increased**
  - Overfit subset of the data — **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly" — **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization — **Try with and without. Test also Layer norm, group norm**
- Regularization strategies — **Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
- Optimization strategy

Optimization review

- https://www.deeplearning.ai/ai-notes/optimization/index.html

- On gradient based optimization methods:

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

- Bottou, L., Curtis, F. E., & Nocedal, J. (2018). Optimization methods for large-scale machine learning. SIAM Review

- Sun, R.-Y. (2020). Optimization for deep learning: An overview. Journal of the Operations Research Society of China

Optimization

Optimization
Overview

Goal of optimization in ML:

Minimize cost over batch: $\sum_{i=1}^{N} L^{(i)}$

where $L^{(i)}$ is the loss $L(y^{(i)}, \hat{y}^{(i)})$ of the i-th training example of batch

Want the optimization to:

- Converge quickly
- Find a good local minima (or even global minima)

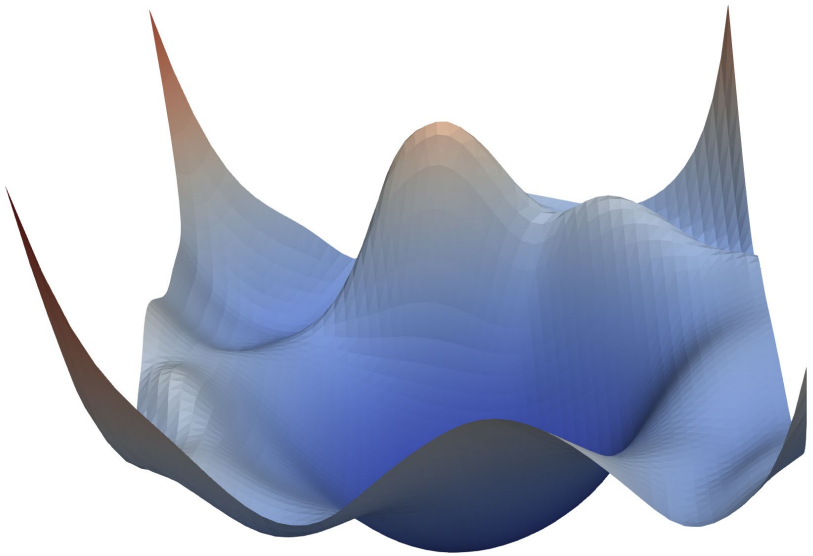
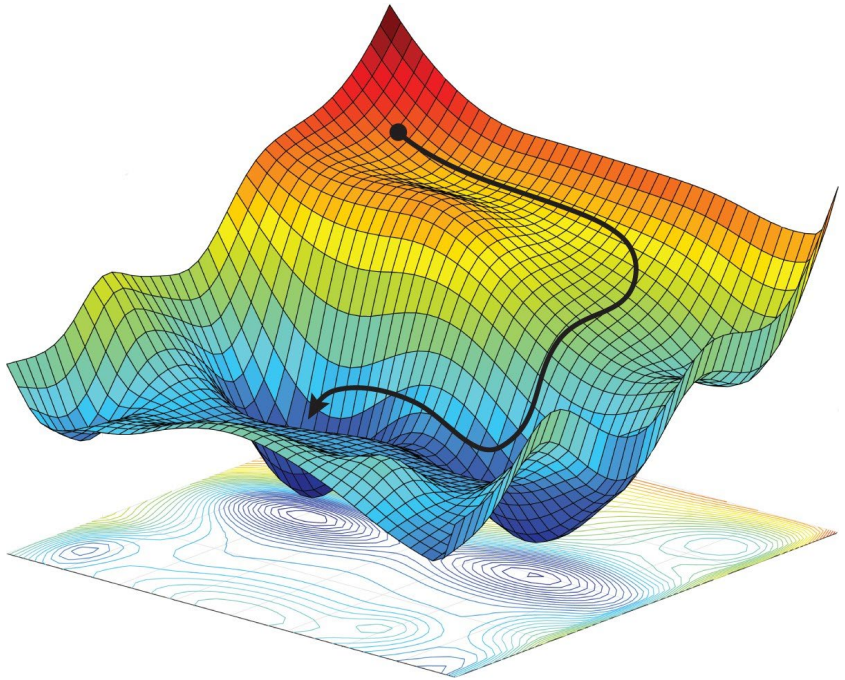


Image credit: https://www.cs.umd.edu/~tomg/projects/landscapes/

Optimization
Overview - for NN

Gradient descent (and variants) is the preferred way to optimize neural networks

Choice of optimizer and hyper-parameters affect speed of convergence and kind of local minima found



A. Amini et al. Spatial Uncertainty Sampling for End-to-End Control, 2019

Optimization
Cost Function - Disambiguation

In this class:

J : cost function

=> average of loss over a single iteration

In ML literature:

- loss function L
- cost function J
- error function E

are sometimes used interchangeably, and

sometimes used like they are in this class

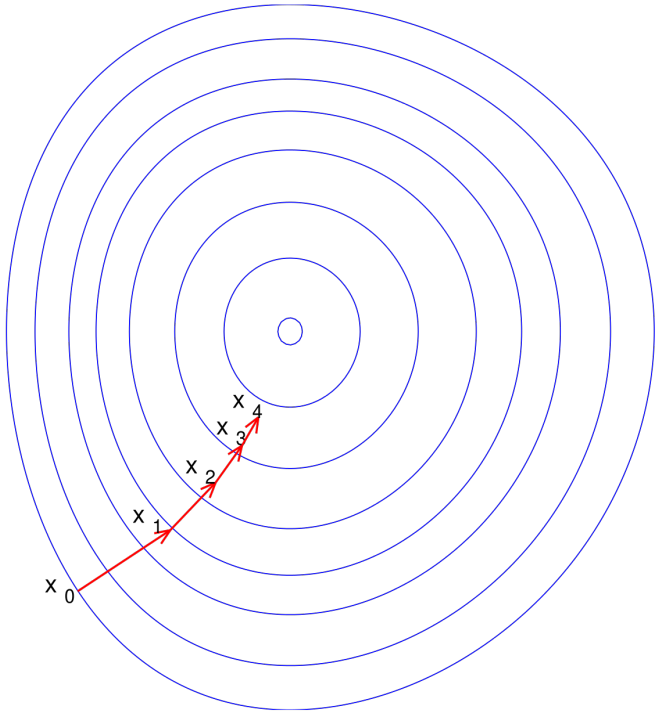
Optimization
Gradient descent

1. Compute J
2. Find $\frac{\partial J}{\partial \mathbf{W}}$ (equivalent notation: $\nabla_{\mathbf{W}} J$)
3. Update parameters with:

- $\mathbf{W} := \mathbf{W} - \alpha \frac{\partial J}{\partial \mathbf{W}}$

  Notation:

  - $\mathbf{W}$ = all parameters of model $(\mathbf{W}^{[1]}, \mathbf{b}^{[1]}, \ldots, \mathbf{W}^{[n]}, \mathbf{b}^{[n]})$
  - α = learning rate



Optimization
Gradient descent

For a training set with N examples:

(Vanilla / Batch) Gradient descent (GD):

- $J = \frac{1}{N}\sum_{i=1}^{N} L^{(i)}$

  where $L^{(i)}$ is the loss $L(y^{(i)}, \hat{y}^{(i)})$ of the i-th example of the training set

- Weights are updated only after calculating the gradient over the entire dataset
  - slow
  - requires large memory

Stochastic gradient descent (SGD):

- $J = L^{(i)}$

  where $L^{(i)}$ is a single example from the training set

- Weights are updated after calculating the gradient of a single example
  - frequent updates, faster convergence
  - requires much less memory than GD
  - can potentially find new, better minima
  - high variance in parameter updates

Optimization
Gradient descent

Q: Can we compromise between vanilla GD and SGD?

Optimization
Gradient descent

Q: Can we compromise between vanilla GD and SGD?

A: Yes! Mini-batch gradient descent:

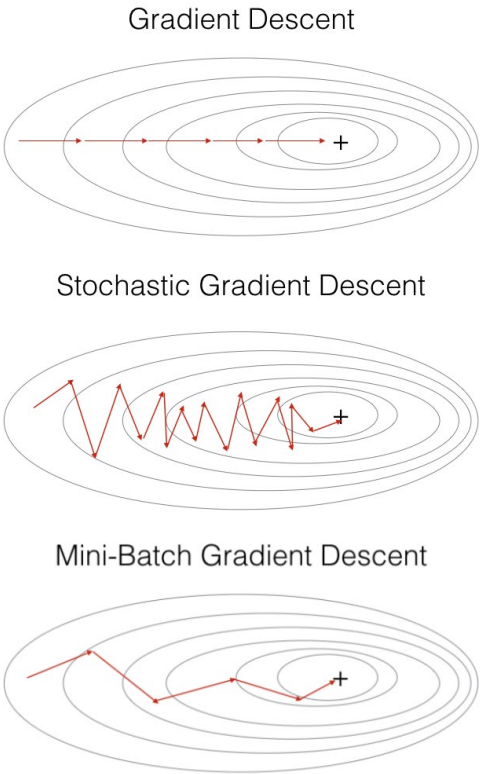
- **Take batches of n_b examples from training set.**

- $J = \frac{1}{n_b}\sum_{i=1}^{n_b} L^{(i)}$,

  where $L^{(i)}$ is the loss $L(y^{(i)}, \hat{y}^{(i)})$ of the i-th example of the mini-batch

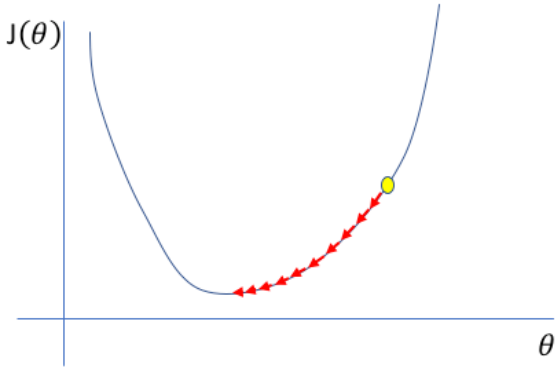
- Weights are updated after every mini-batch:
  - faster convergence than GD
  - reduces variance of parameter updates compared to SGD

    → more stable convergence



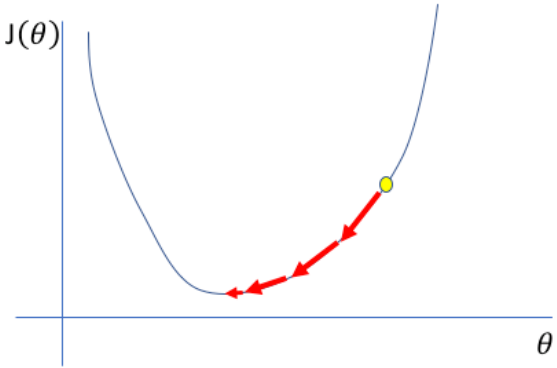
Optimization
Learning rate

Too low



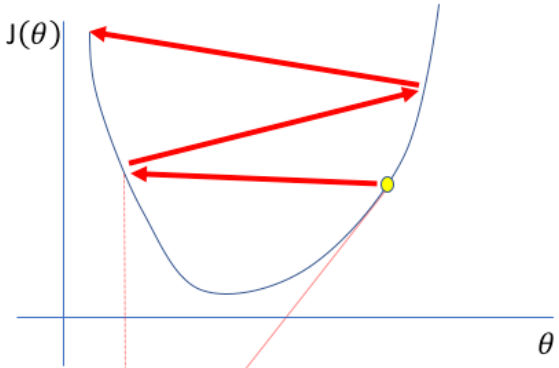
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



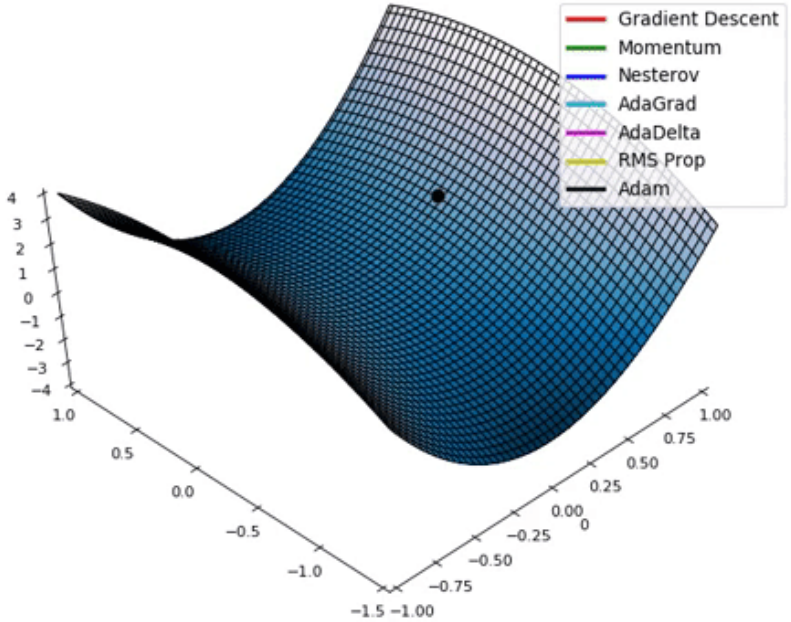
Too large of a learning rate causes drastic updates which lead to divergent behaviors

Image credit: Jeremy Jordan (https://www.jeremyjordan.me/nn-learning-rate/)

Optimization
Optimizers

Variants of gradient descent are commonly used in practice to speed-up and improve convergence:

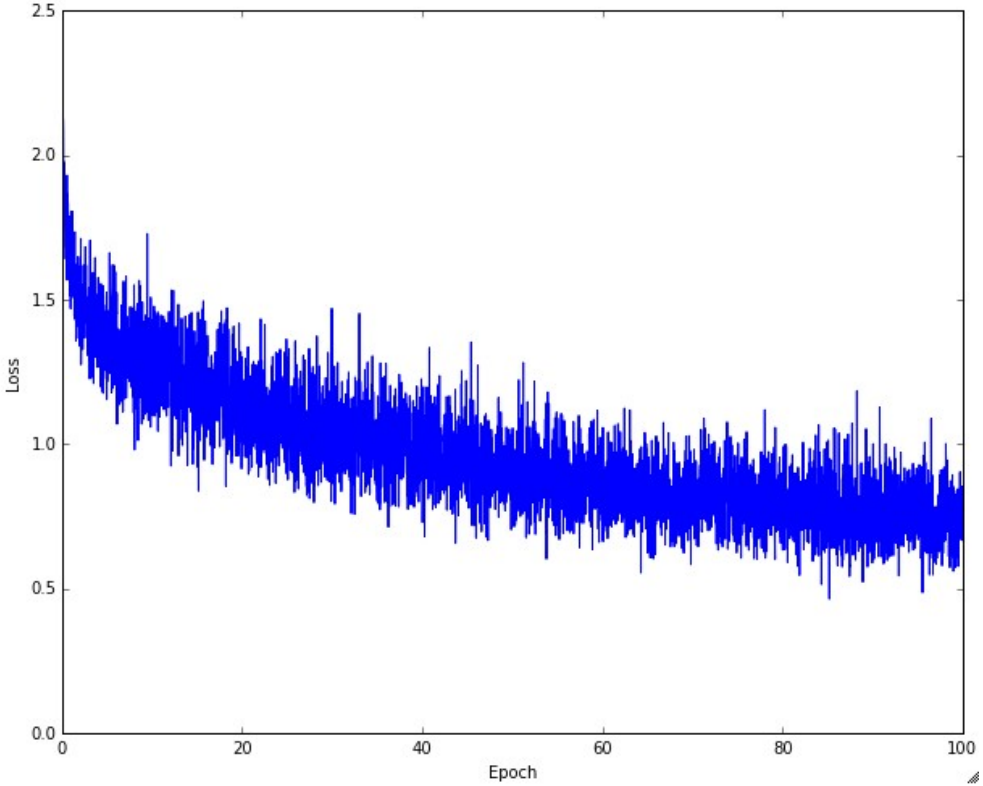
- Momentum update
- Nesterov Accelerated Gradient (NAG)
- Adagrad
- Adadelta
- RMSprop
- Adam
- and more…

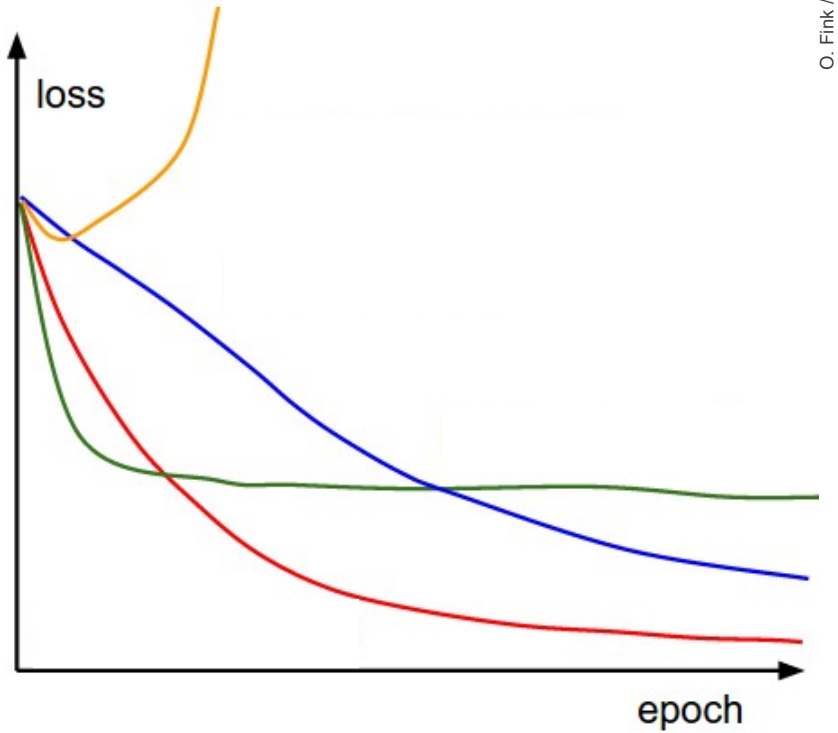


NN TIPS & TRICKS

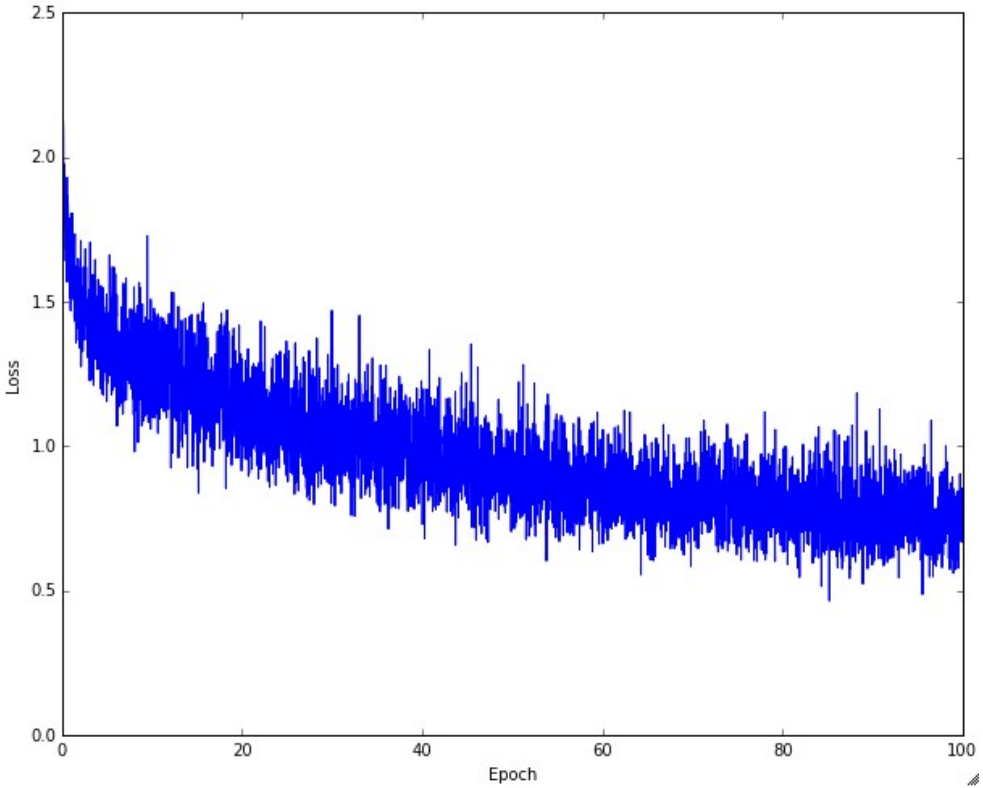
- Preprocess data
  - **Normalize each dimension of the input**
- Design an architecture
  - **Start from state-of-the-art designs**
  - Loss function
    - **Select task appropriate loss function (pytorch losses)**
  - Layers type
    - **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters
    - **Start with popular designs**
  - Activation function
    - **ReLu, elu**
- Sanity check
  - **Only once**
  - Gradient check
    - **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics
    - **At chance or when regularization is increased**
  - Overfit subset of the data
    - **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly"
  - **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization
  - **Try with and without. Test also Layer norm, group norm**
- Regularization strategies
  - **Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
- Optimization strategy
  - **AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
- Monitor learning process

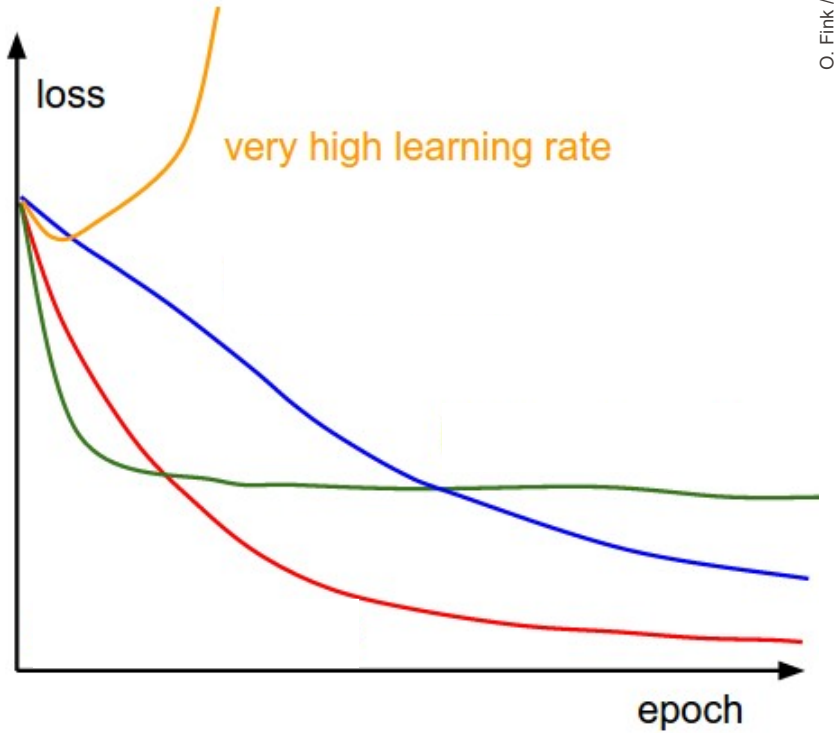
Monitor and visualize the loss curve



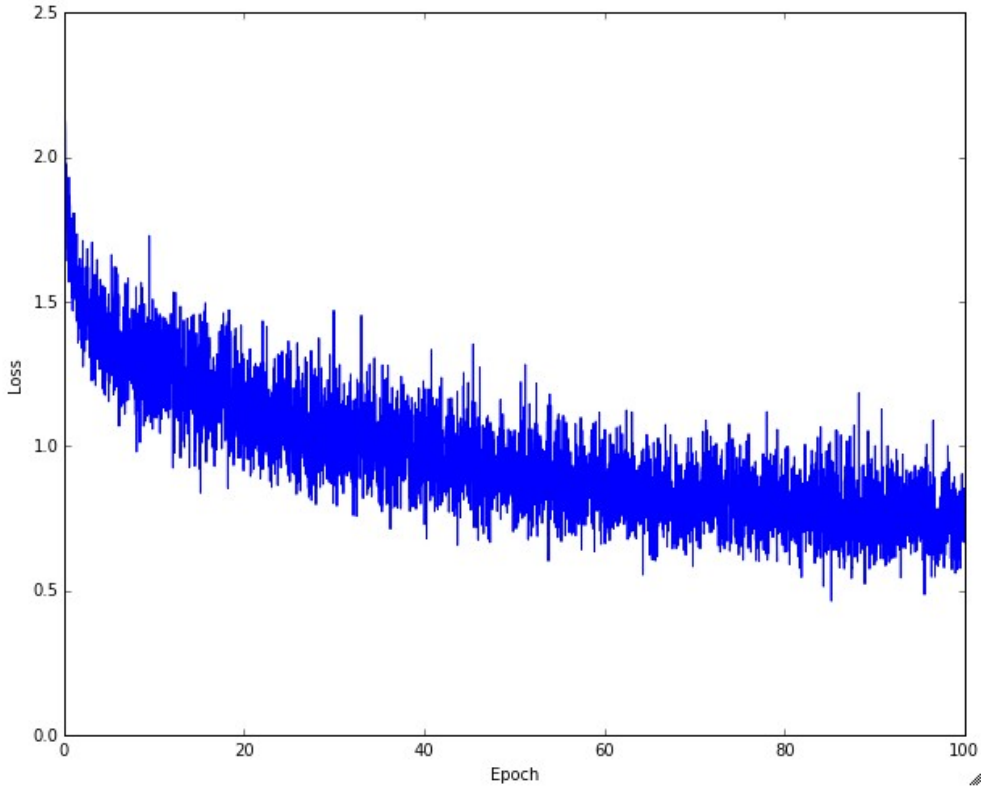


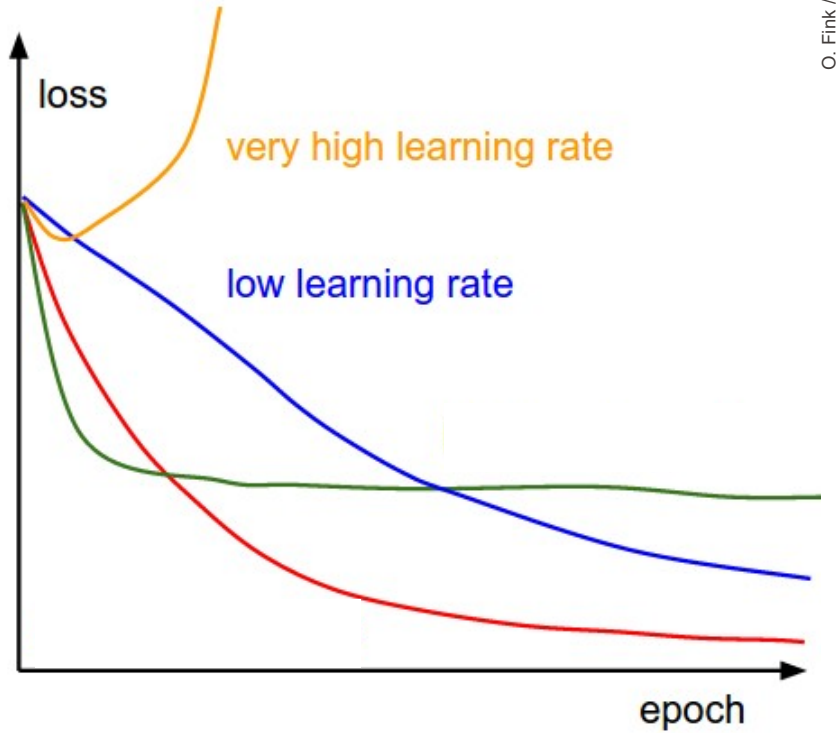
Monitor and visualize the loss curve



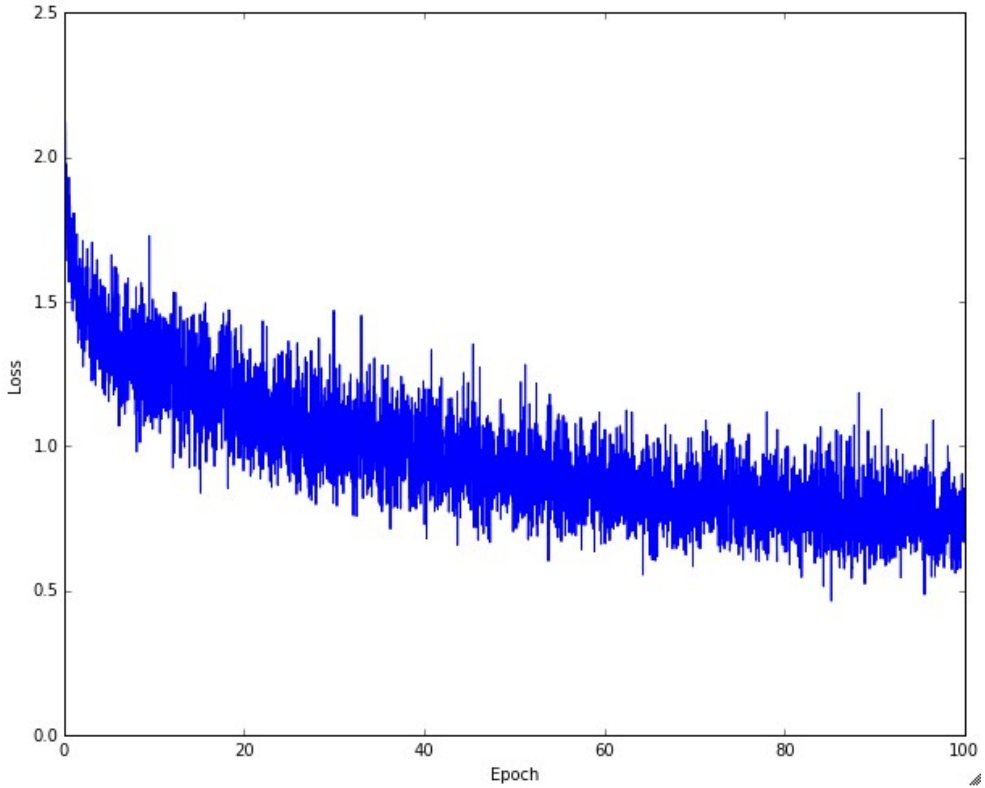


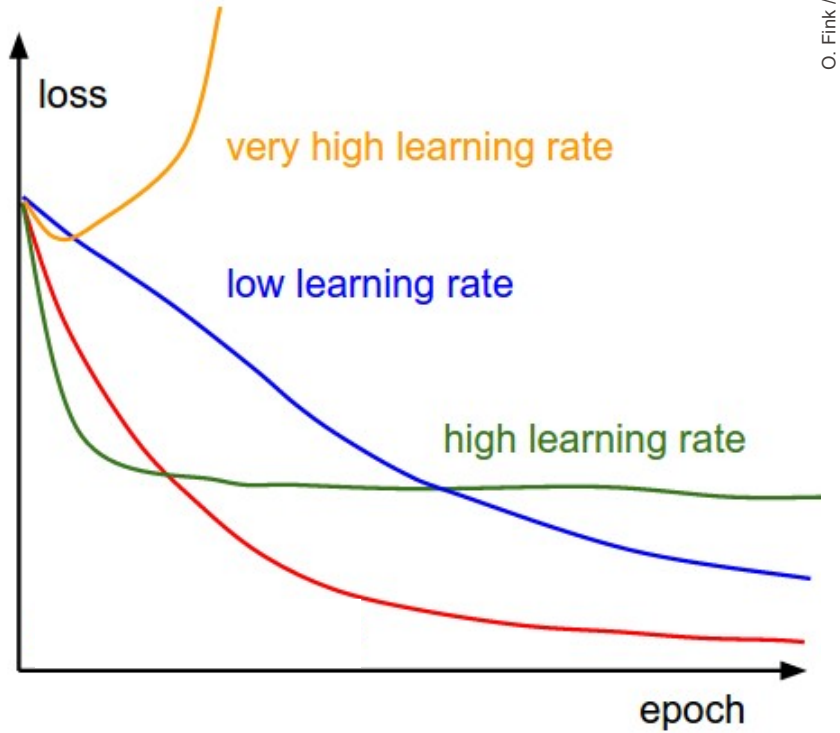
Monitor and visualize the loss curve





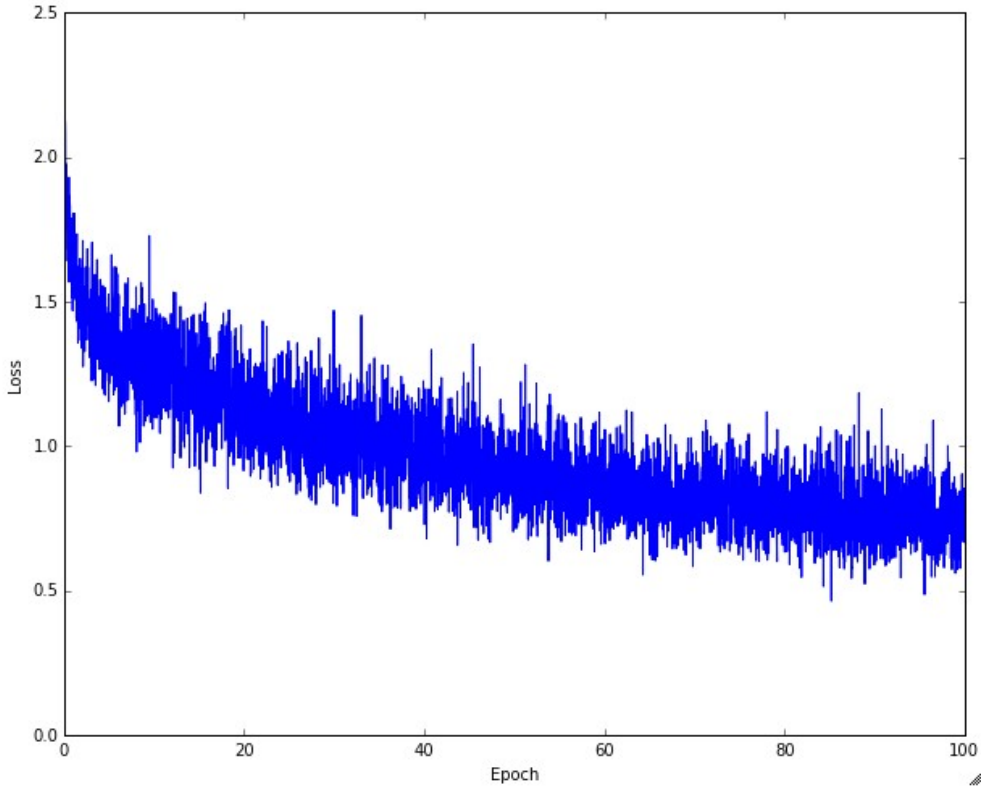
Monitor and visualize the loss curve

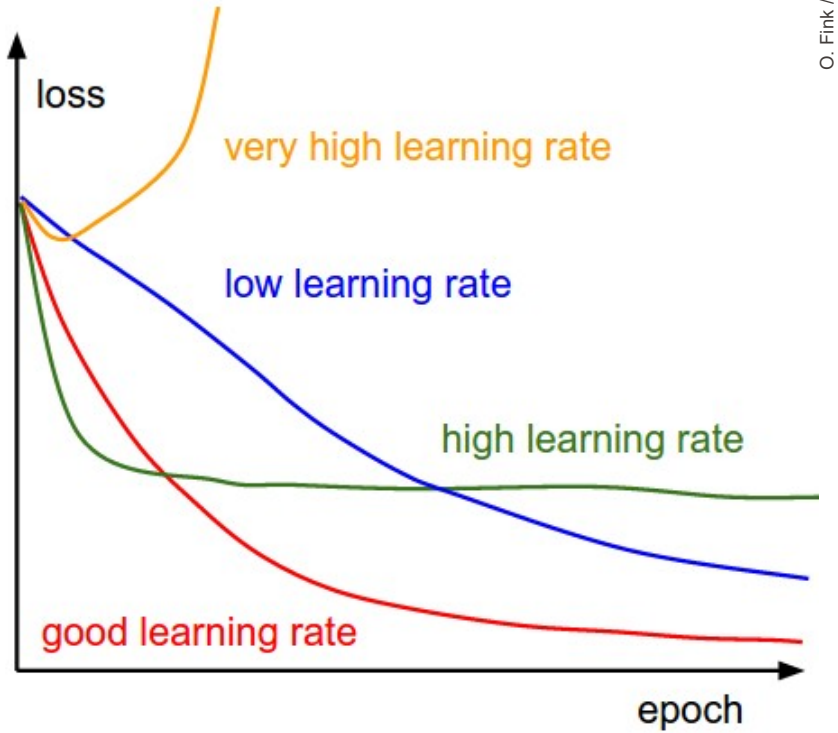


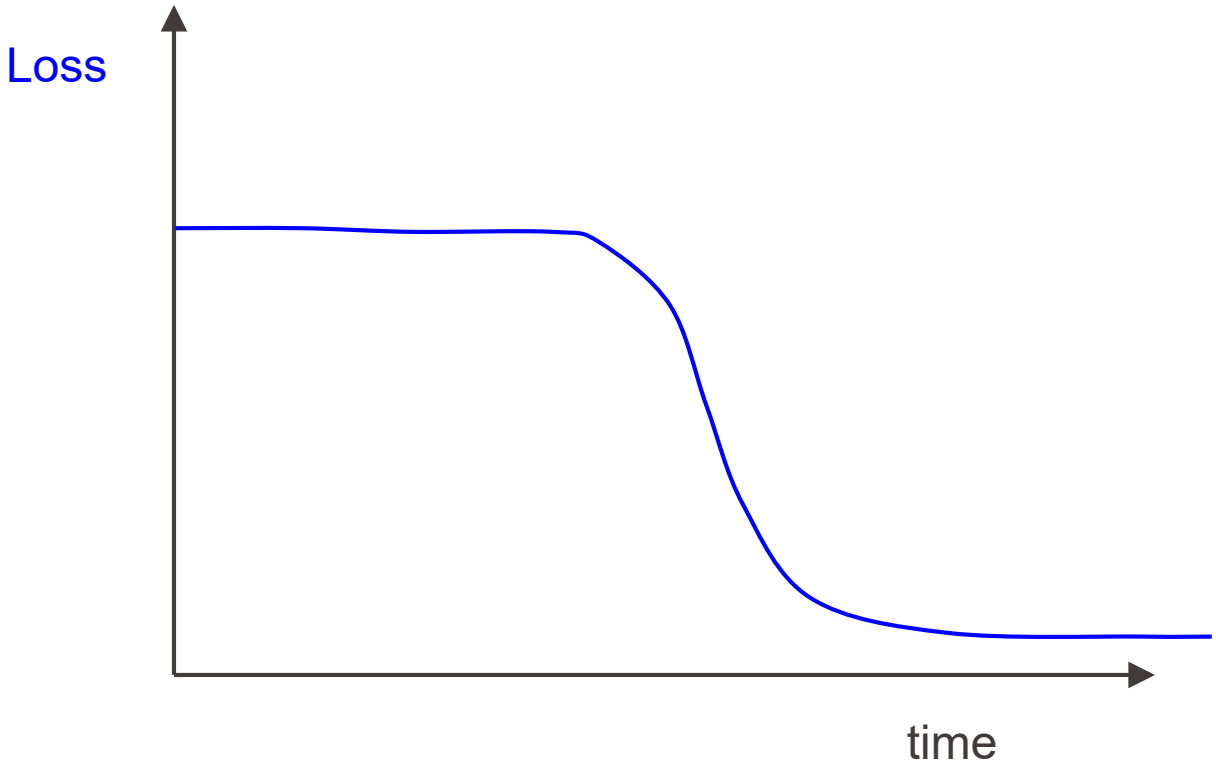


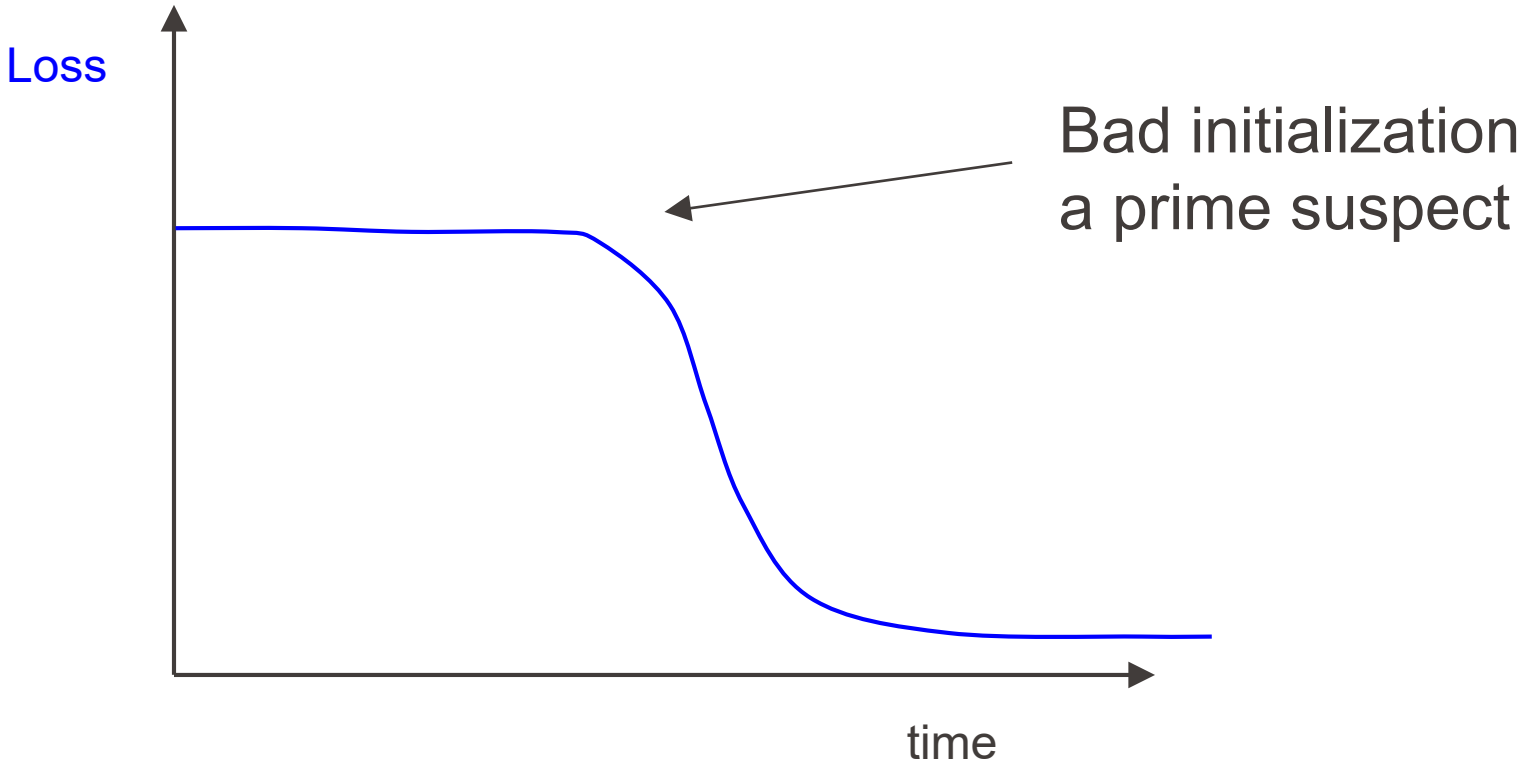
O. Fink / A. Alahi

Monitor and visualize the loss curve

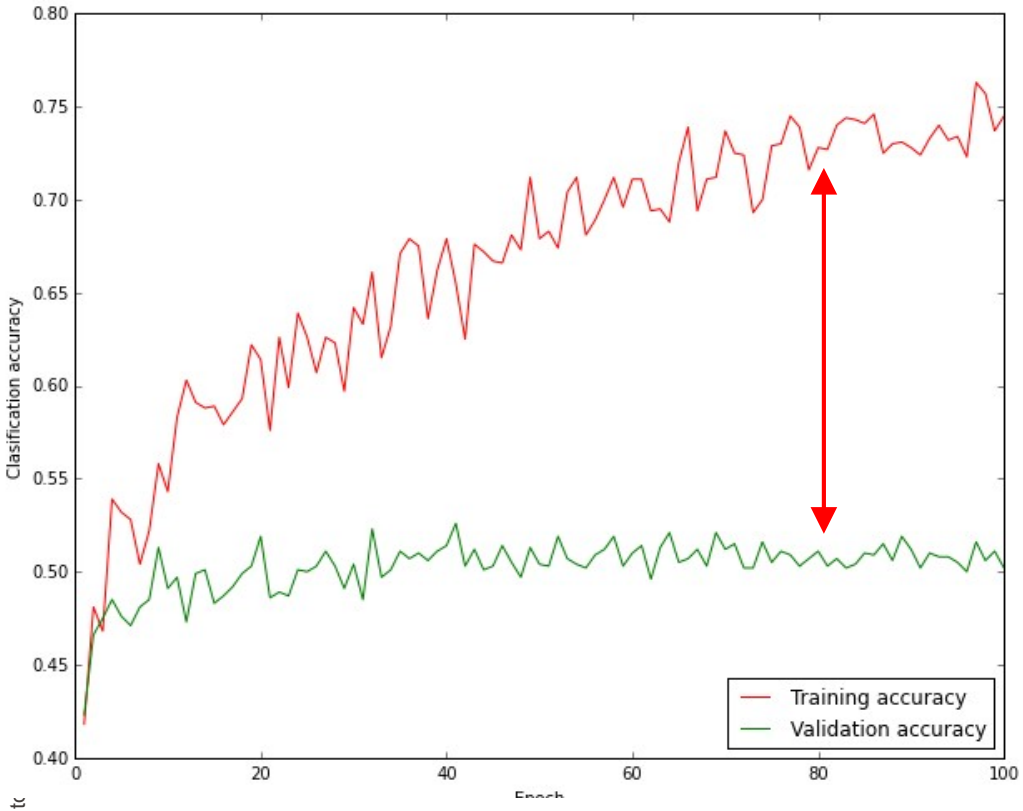





Loss
time


Loss
Bad initialization
a prime suspect
time

Monitor and visualize the accuracy:



big gap = overfitting

=> increase regularization strength?

no gap

=> increase model capacity?

Track the ratio of weight updates / weight magnitudes:

```
# assume parameter vector W and its gradient vector dW
param_scale = np.linalg.norm(W.ravel())
update = -learning_rate*dW # simple SGD update
update_scale = np.linalg.norm(update.ravel())
W += update # the actual update
print update_scale / param_scale # want ~1e-3
```

Ratio between the updates and values: ~ 0.0002 / 0.02 = 0.01 (about okay)

want this to be somewhere around 0.001 or so

NN TIPS & TRICKS

- Preprocess data — **Normalize each dimension of the input**
- Design an architecture — **Start from state-of-the-art designs**
  - Loss function — **Select task appropriate loss function (pytorch losses)**
  - Layers type — **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters — **Start with popular designs**
  - Activation function — **ReLu, elu**
- Sanity check — **Only once**
  - Gradient check — **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics — **At chance or when regularization is increased**
  - Overfit subset of the data — **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly" — **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization — **Try with and without. Test also Layer norm, group norm**
- Regularization strategies — **Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
- Optimization strategy — **AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
- Monitor learning process — **Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**

NN TIPS & TRICKS

- Preprocess data — **Normalize each dimension of the input**
- Design an architecture — **Start from state-of-the-art designs**
  - Loss function — **Select task appropriate loss function (pytorch losses)**
  - Layers type — **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters — **Start with popular designs**
  - Activation function — **ReLu, elu**
- Sanity check — **Only once**
  - Gradient check — **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics — **At chance or when regularization is increased**
  - Overfit subset of the data — **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly" — **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization — **Try with and without. Test also Layer norm, group norm**
- Regularization strategies — **Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
- Optimization strategy — **AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
- Monitor learning process — **Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**
- Hyperparameters search

Cross-validation strategy

- Coarse -> fine cross-validation in stages
- First stage: only a few epochs to get rough idea of what params work
  Second stage: longer running time, finer search
  … (repeat as necessary)

- Tip for detecting explosions in the solver:

If the cost is ever > 3 * original cost, break out early

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning_rate=lr, verbose=False)
```

note it's best to optimize in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

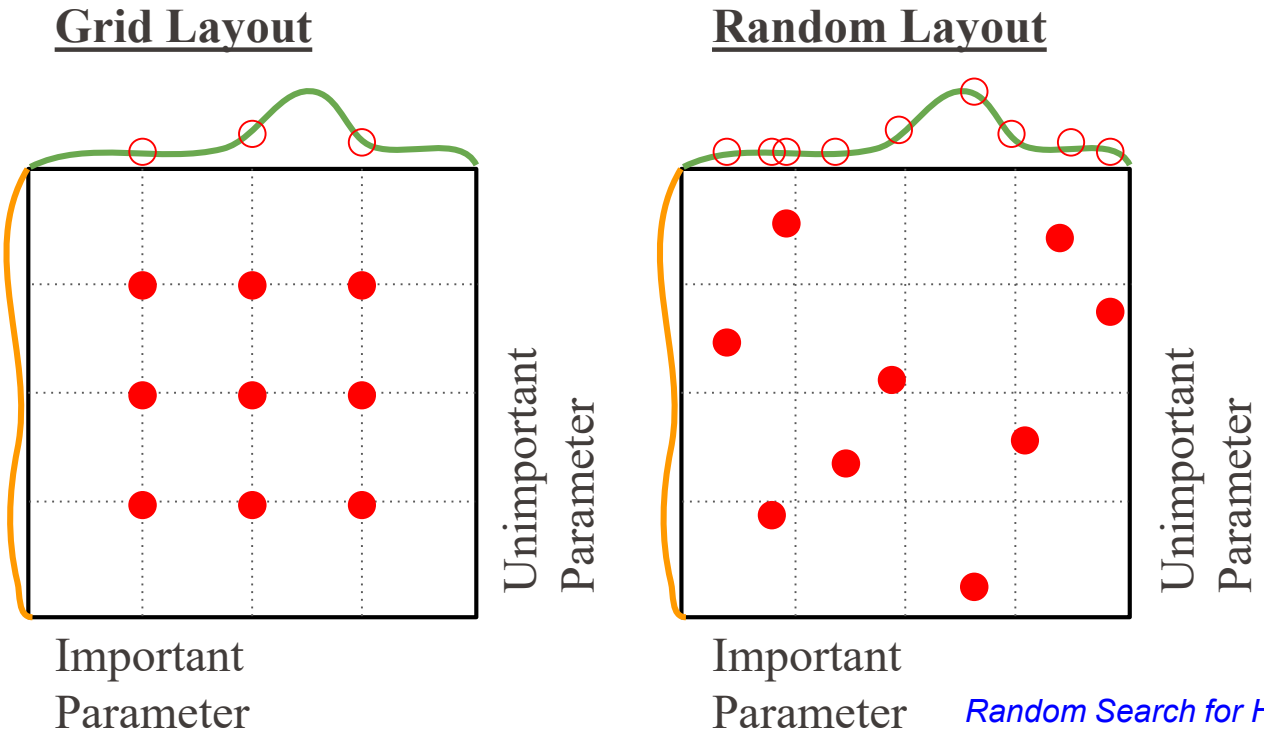
adjust range

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons.

Random Search vs. Grid Search

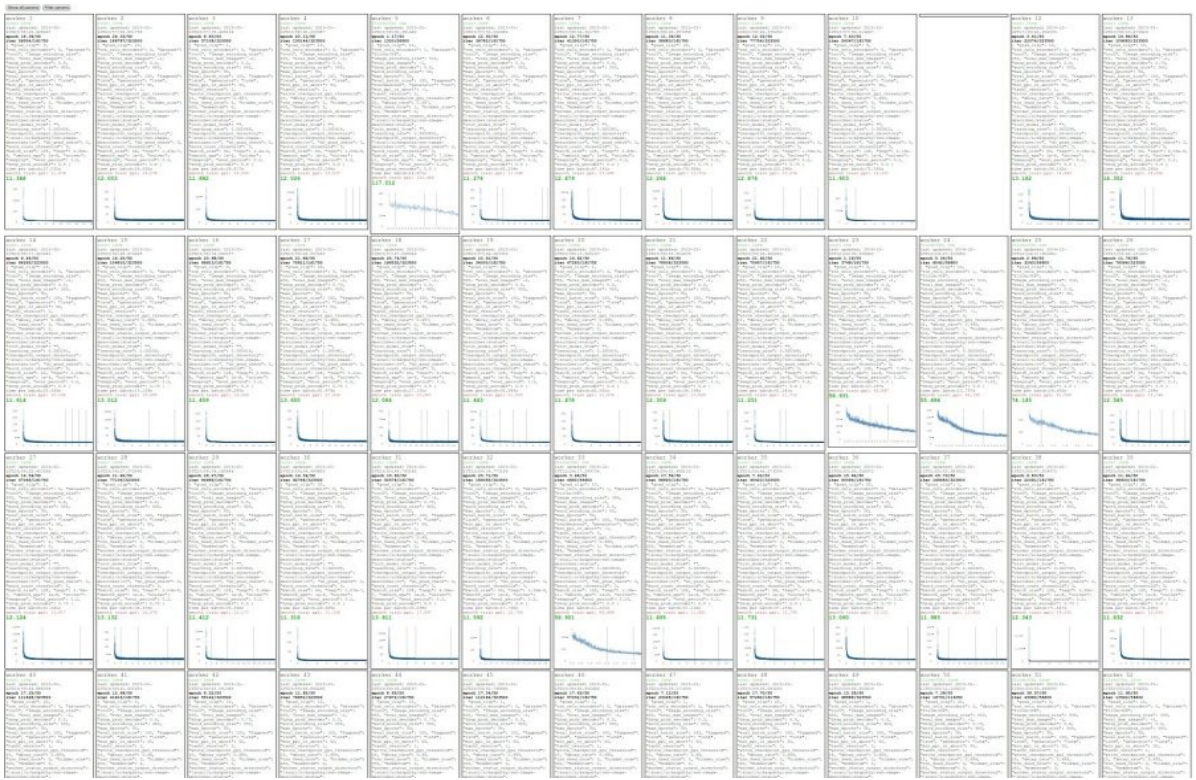


Random Search for Hyper-Parameter Optimization
Bergstra and Bengio, 2012

Hyperparameters to play with

- network architecture
- learning rate, its decay schedule, update type
- regularization (L2/Dropout strength)

Cross-validation “command center”



NN TIPS & TRICKS

▪ Preprocess data	**Normalize each dimension of the input**
▪ Design an architecture	**Start from state-of-the-art designs**
• Loss function	**Select task appropriate loss function (pytorch losses)**
• Layers type	**“Conv” for signals such as images, and more recently Transformers**
• #layers, #Filters	**Start with popular designs**
• Activation function	**ReLu, elu**
▪ Sanity check	**Only once**
• Gradient check	**Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
• Loss dynamics	**At chance or when regularization is increased**
• Overfit subset of the data	**Use small subset of the data (while regularization is off)**
• Initialize weight “smartly”	**Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
• Batch normalization	**Try with and without. Test also Layer norm, group norm**
• Regularization strategies	**Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
▪ Optimization strategy	**AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
▪ Monitor learning process	**Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**
▪ Hyperparameters search	**Random sample in log space (wide to coarse) (e.g., ray autotuner)**

NN TIPS & TRICKS

- Preprocess data — **Normalize each dimension of the input**
- Design an architecture — **Start from state-of-the-art designs**
  - Loss function — **Select task appropriate loss function (pytorch losses)**
  - Layers type — **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters — **Start with popular designs**
  - Activation function — **ReLu, elu**
- Sanity check — **Only once**
  - Gradient check — **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics — **At chance or when regularization is increased**
  - Overfit subset of the data — **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly" — **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization — **Try with and without. Test also Layer norm, group norm**
- Regularization strategies — **Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
- Optimization strategy — **AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
- Monitor learning process — **Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**
- Hyperparameters search — **Random sample in log space (wide to coarse) (e.g., ray autotuner)**
- Model ensembles

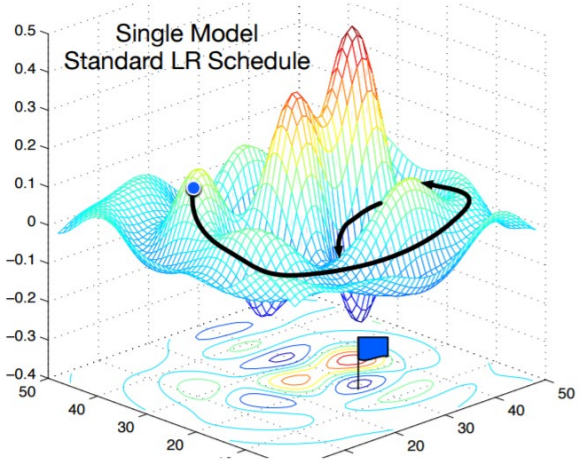
Model Ensembles

1. Train multiple independent models
2. At test time average their results

- Enjoy 2% extra performance

Model Ensembles: Tips and Tricks

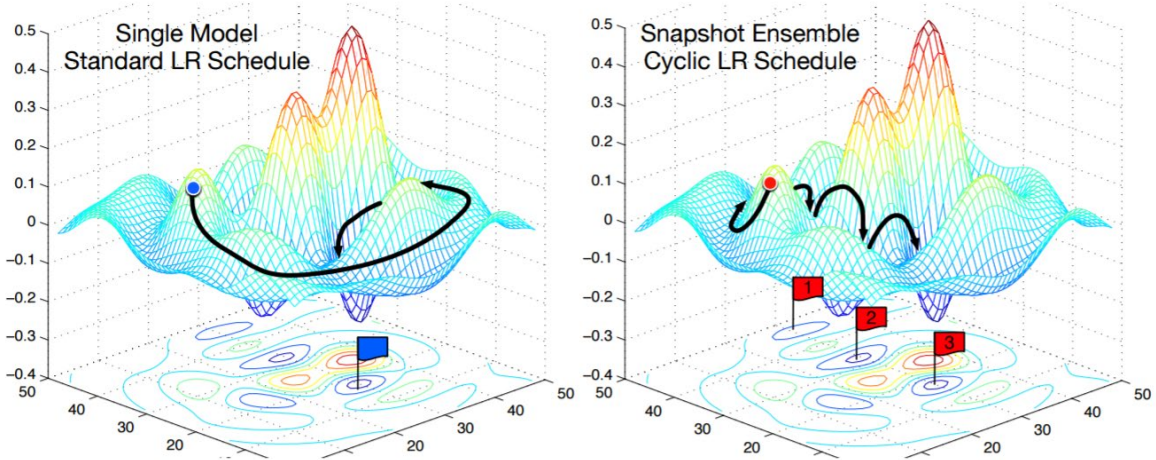
- Instead of training independent models, use multiple snapshots of a single model during training!



Loshchilov and Hutter, “SGDR: Stochastic gradient descent with restarts”, arXiv 2016
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

Model Ensembles: Tips and Tricks

- Instead of training independent models, use multiple snapshots of a single model during training!



Loshchilov and Hutter, “SGDR: Stochastic gradient descent with restarts”, arXiv 2016
Huang et al, “Snapshot ensembles: train 1, get M for free”, ICLR 2017
Figures copyright Yixuan Li and Geoff Pleiss, 2017. Reproduced with permission.

NN TIPS & TRICKS

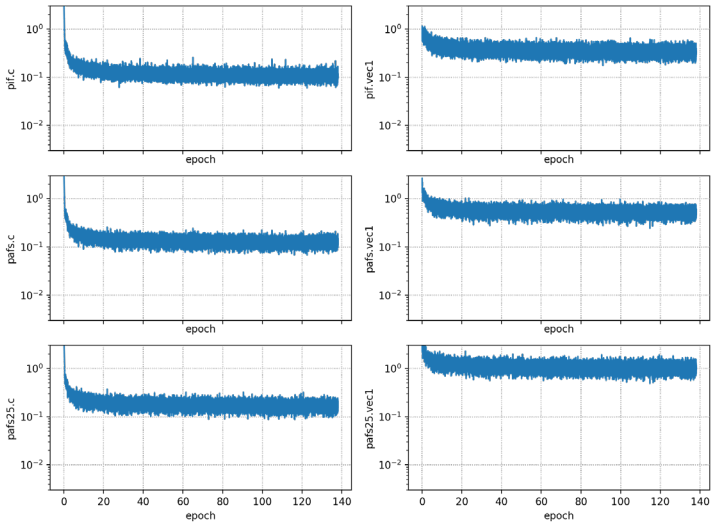
▪ Preprocess data	**Normalize each dimension of the input**
▪ Design an architecture	**Start from state-of-the-art designs**
• Loss function	**Select task appropriate loss function (pytorch losses)**
• Layers type	**"Conv" for signals such as images, and more recently Transformers**
• #layers, #Filters	**Start with popular designs**
• Activation function	**ReLu, elu**
▪ Sanity check	**Only once**
• Gradient check	**Analytical VS num grad (while turning all regularization off)** ***(fix rand seed)***
• Loss dynamics	**At chance or when regularization is increased**
• Overfit subset of the data	**Use small subset of the data (while regularization is off)**
• Initialize weight "smartly"	**Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
• Batch normalization	**Try with and without. Test also Layer norm, group norm**
• Regularization strategies	**Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
▪ Optimization strategy	**AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
▪ Monitor learning process	**Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**
▪ Hyperparameters search	**Random sample in log space (wide to coarse) (e.g., ray autotuner)**
▪ Model ensembles	**Average top 10 performing models**

NN TIPS & TRICKS

▪ Preprocess data	**Normalize each dimension of the input**
▪ Design an architecture	**Start from state-of-the-art designs**
• Loss function	**Select task appropriate loss function (pytorch losses)**
• Layers type	**“Conv” for signals such as images, and more recently Transformers**
• #layers, #Filters	**Start with popular designs**
• Activation function	**ReLu, elu**
▪ Sanity check	**Only once**
• Gradient check	**Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
• Loss dynamics	**At chance or when regularization is increased**
• Overfit subset of the data	**Use small subset of the data (while regularization is off)**
• Initialize weight “smartly”	**Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
• Batch normalization	**Try with and without. Test also Layer norm, group norm**
• Regularization strategies	**Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
▪ Optimization strategy	**AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
▪ Monitor learning process	**Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**
▪ Hyperparameters search	**Random sample in log space (wide to coarse) (e.g., ray autotuner)**
▪ Model ensembles	**Average top 10 performing models**
▪ Sub-loss magnitude ~ 1	**Numerical stability**

Multi-Task Losses

- Often necessary to do multiple tasks simultaneously. Below are training curves showing the value of their sub-losses for 6 sub-tasks.
- Each sub-task can be multiplied by a constant, the “weight” of the sub-task.
- Sometimes, there is no natural weight. Then, a good initial weight guess is to make every sub-loss ~1.



NN TIPS & TRICKS

▪ Preprocess data	**Normalize each dimension of the input**
▪ Design an architecture	**Start from state-of-the-art designs**
• Loss function	**Select task appropriate loss function (pytorch losses)**
• Layers type	**“Conv” for signals such as images, and more recently Transformers**
• #layers, #Filters	**Start with popular designs**
• Activation function	**ReLu, elu**
▪ Sanity check	**Only once**
• Gradient check	**Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
• Loss dynamics	**At chance or when regularization is increased**
• Overfit subset of the data	**Use small subset of the data (while regularization is off)**
• Initialize weight “smartly”	**Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
• Batch normalization	**Try with and without. Test also Layer norm, group norm**
• Regularization strategies	**Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
▪ Optimization strategy	**AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
▪ Monitor learning process	**Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**
▪ Hyperparameters search	**Random sample in log space (wide to coarse) (e.g., ray autotuner)**
▪ Model ensembles	**Average top 10 performing models**
▪ Sub-loss magnitude ~ 1	**Numerical stability**
▪ Pretrained networks	**Especially helpful when own training data is small**

NN TIPS & TRICKS

- “You need a lot of a data if you want to train/use Deep NNs”

IMAGENET

www.image-net.org

22K categories and **14M** images

- Animals
  - Bird
  - Fish
  - Mammal
  - Invertebrate
- Plants
  - Tree
  - Flower
- Food
- Materials
- Structures
- Artifact
  - Tools
  - Appliances
  - Structures
- Person
- Scenes
  - Indoor
  - Geological Formations
- Sport Activities

Deng, Dong, Socher, Li, Li, & Fei-Fei, 2009

IMAGENET Large Scale Visual Recognition Challenge

Steel drum

Output:

Scale

T-shirt

Steel drum

Drumstick

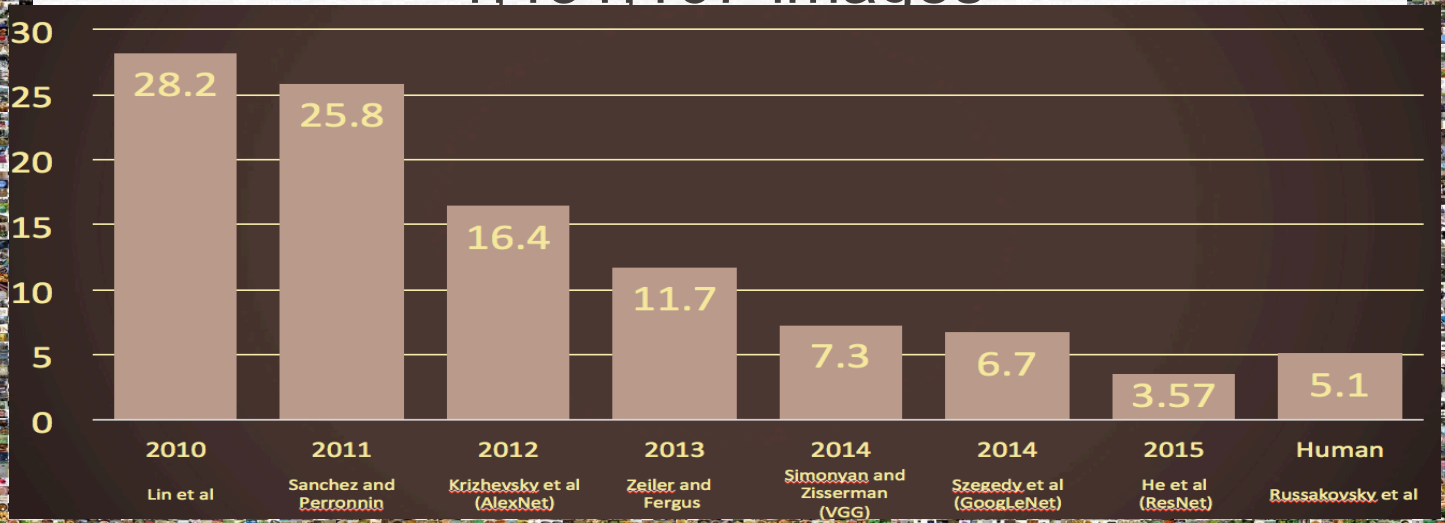
Mud turtle

IMAGENET Large Scale Visual Recognition Challenge

The Image Classification Challenge:

1,000 object classes

1,431,167 images



Russakovsky et al., 2014

NN TIPS & TRICKS

- “You need a lot of a data if you want to train/use Deep NNs”

- What if you don’t?
- Transfer Learning

Transfer Learning with CNNs

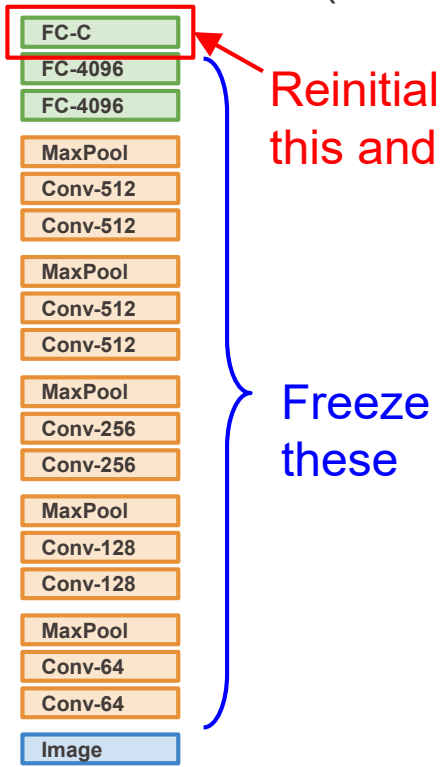
1. Train on Imagenet

Transfer Learning with CNNs

O. Fink / A. Alahi

1. Train on Imagenet

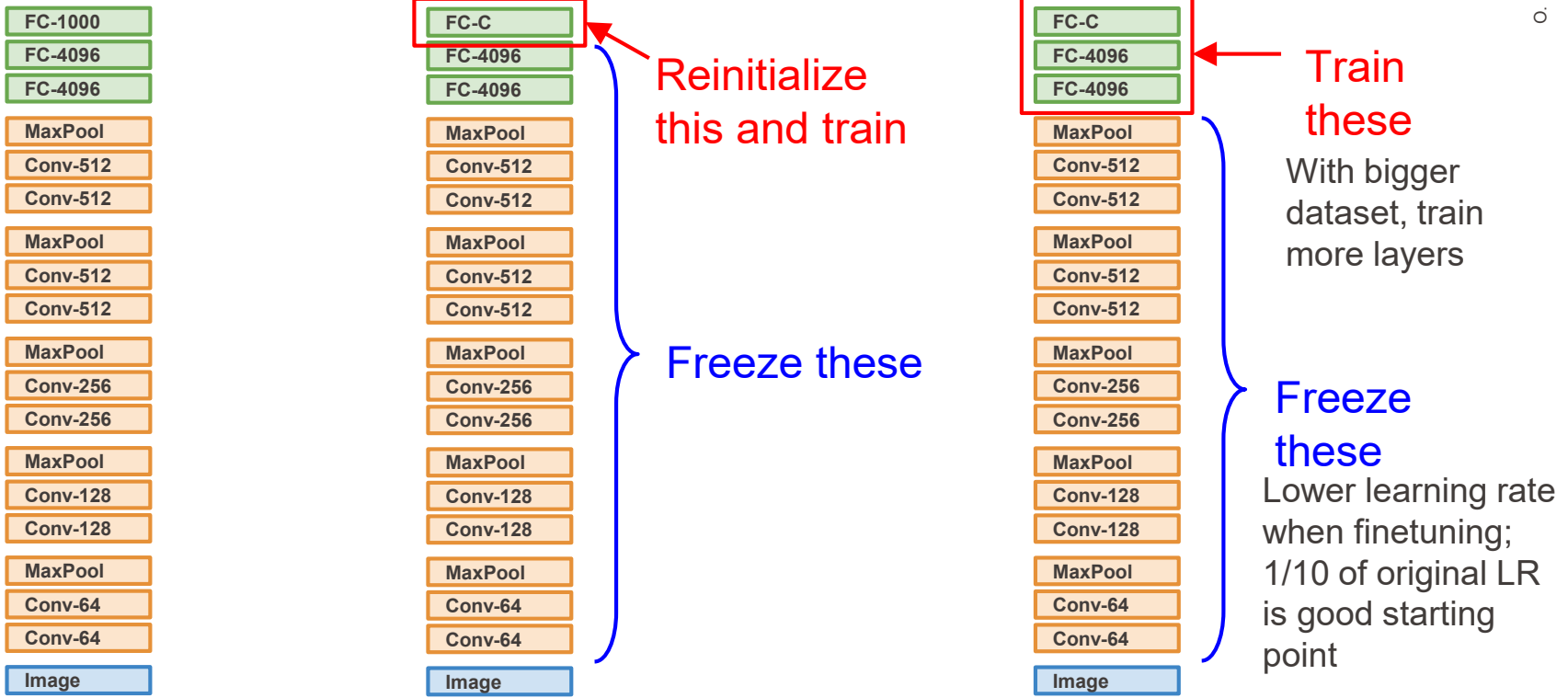
2. Small Dataset (C classes)

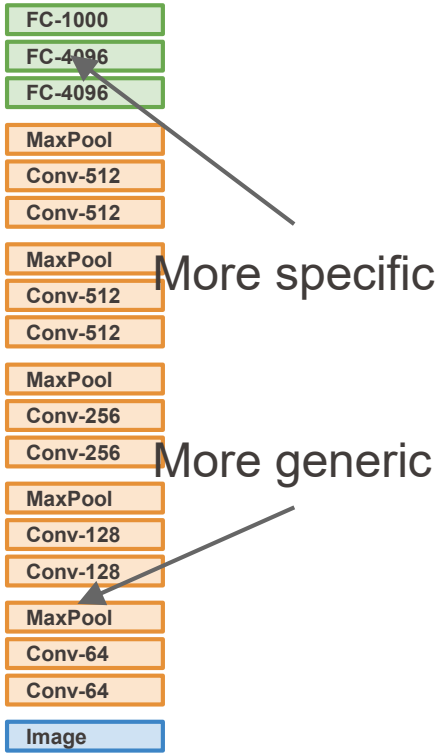


Reinitialize this and train

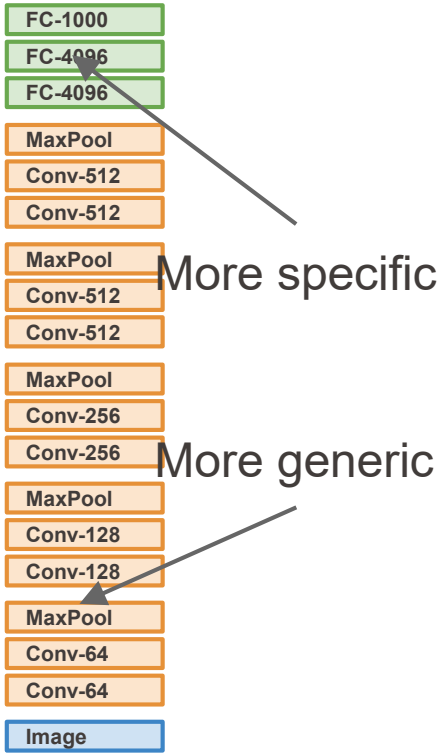
Freeze these

Transfer Learning with CNNs

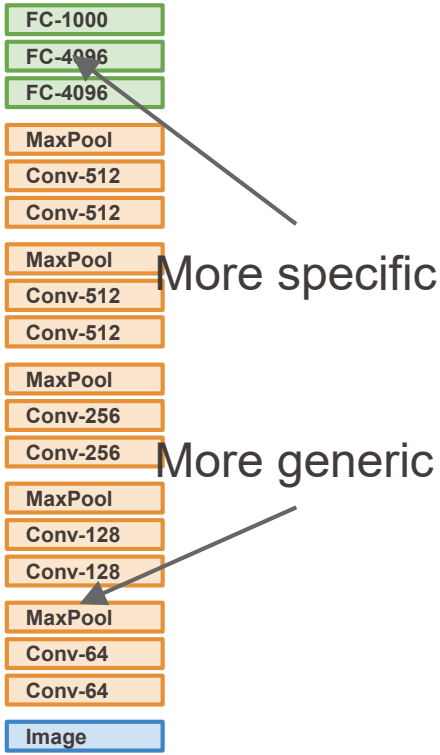




	very similar dataset	very different dataset
very little data	?	?
quite a lot of data	?	?



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	?
quite a lot of data	Finetune a few layers	?



	very similar dataset	very different dataset
very little data	Use Linear Classifier on top layer	You're in trouble… Try linear classifier from different stages
quite a lot of data	Finetune a few layers	Finetune a larger number of layers

Takeaway

- When you have some a small dataset

1. Find a very large dataset that has similar data, train a big network there
2. Transfer learn to your dataset

- Deep learning frameworks provide a “Model Zoo” of pretrained models so you don’t need to train your own
- PyTorch: https://github.com/pytorch/vision

- https://teachablemachine.withgoogle.com/train/image

NN TIPS & TRICKS

- Preprocess data — **Normalize each dimension of the input**
- Design an architecture — **Start from state-of-the-art designs**
  - Loss function — **Select task appropriate loss function (pytorch losses)**
  - Layers type — **"Conv" for signals such as images, and more recently Transformers**
  - #layers, #Filters — **Start with popular designs**
  - Activation function — **ReLu, elu**
- Sanity check — **Only once**
  - Gradient check — **Analytical VS num grad (while turning all regularization off) *(fix rand seed)***
  - Loss dynamics — **At chance or when regularization is increased**
  - Overfit subset of the data — **Use small subset of the data (while regularization is off)**
- Initialize weight "smartly" — **Xavier (Glorot) / Kaiming (He) initialization (pytorch init)**
- Batch normalization — **Try with and without. Test also Layer norm, group norm**
- Regularization strategies — **Data augmentation, L1, L2, dropout (no dropout in ResNet), DropBlock**
- Optimization strategy — **AdamW, Adam, SGD+Nesterov momentum, warm-up, learning rate scheduler**
- Monitor learning process — **Check loss, training/validation accuracy, ratio of weights updates (e.g., Wandb)**
- Hyperparameters search — **Random sample in log space (wide to coarse) (e.g., ray autotuner)**
- Model ensembles — **Average top 10 performing models**
- Sub-loss magnitude ~ 1 — **Numerical stability**
- Pretrained networks — **Especially helpful when own training data is small**

- + Data distribution
- + Gradient flow
- **+ Visualize your input data, intermediate values, results**

References

- Book: Deep Learning by Ian Goodfellow, Yoshua Bengio, Aaron Courville (http://www.deeplearningbook.org/)
- Class on CNN: http://cs231n.stanford.edu/
- Good tuto on gradient check:

http://cs231n.github.io/neural-networks-3/

- On gradient based optimization methods:

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

Additional materials

- CNN demo on CIFAR-10:

http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html