

# Image Classification with Linear Classifiers

# Image Classification

A Core Task in Computer Vision

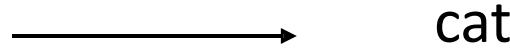
Today:

- The image classification task
- Two basic data-driven approaches to image classification
  - K-nearest neighbor and linear classifier

# Image Classification: A core task in Computer Vision

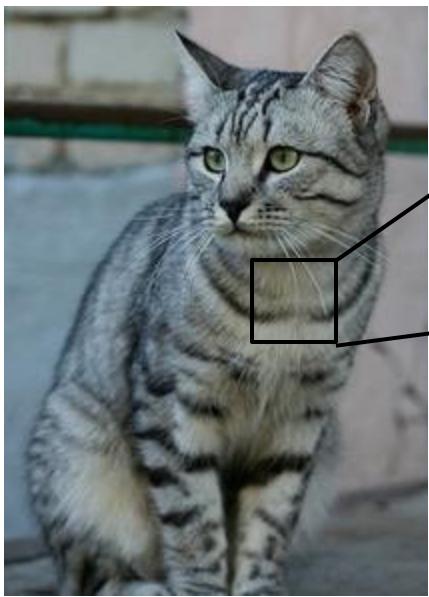


(assume given a set of possible labels)  
{dog, cat, truck, plane, ...}



This image by Nikita is  
licensed under CC-BY 2.0

# The Problem: Semantic Gap



[105 112 188 111 184 99 186 99 96 183 112 119 184 97 93 87]
[ 91 98 182 186 184 79 98 183 99 185 123 136 110 185 94 85]
[ 76 85 98 185 128 185 87 96 95 99 115 112 186 183 99 85]
[ 99 81 93 120 131 127 188 95 98 182 99 96 93 181 94]
[106 91 61 64 69 91 88 85 181 187 189 98 75 84 96 95]
[114 108 85 55 55 69 64 54 64 87 112 129 98 74 84 91]
[133 137 147 183 65 81 88 65 52 54 74 84 182 93 85 82]
[128 137 144 148 109 95 86 78 62 65 63 63 60 73 86 181]
[125 133 148 137 119 121 117 94 65 79 88 65 54 64 72 98]
[127 125 131 147 133 127 126 131 111 96 89 75 61 64 72 84]
[115 114 189 123 150 148 131 118 113 189 108 92 74 65 72 78]
[ 89 93 98 97 188 147 131 118 113 114 113 109 186 95 77 88]
[ 63 77 86 81 77 79 182 123 117 115 117 125 125 130 115 87]
[ 62 65 82 89 78 71 88 101 124 126 119 101 187 114 131 119]
[ 63 65 75 88 89 71 62 81 128 138 135 105 81 98 118 118]
[ 87 65 71 87 106 95 69 45 76 138 126 187 92 94 185 112]
[118 97 82 86 117 123 116 66 41 51 95 93 89 95 182 187]
[164 146 112 88 82 120 124 184 76 48 45 66 88 181 182 189]
[157 170 157 128 93 86 114 134 112 97 69 55 78 82 99 94]
[130 128 134 163 139 188 189 118 121 134 114 87 65 53 69 86]
[128 112 96 117 150 144 128 115 184 187 102 93 87 81 72 79]
[123 107 96 86 83 112 153 149 122 189 184 75 80 107 112 99]
[122 121 102 88 82 86 94 117 145 148 153 182 58 78 92 107]
[122 164 148 183 71 56 78 83 93 103 119 139 182 61 69 84]

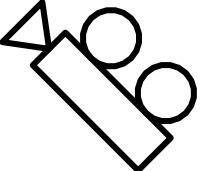
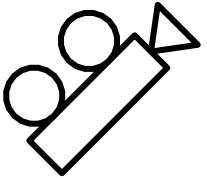
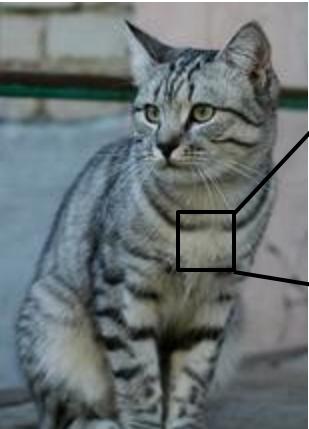
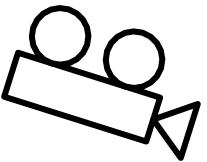
What the computer sees

An image is a tensor of integers  
between [0, 255]:

e.g. 800 x 600 x 3  
(3 channels RGB)

This image by [Nikita](#) is  
licensed under [CC-BY 2.0](#)

# Challenges: Viewpoint variation



All pixels change when  
the camera moves!

# Challenges: Illumination



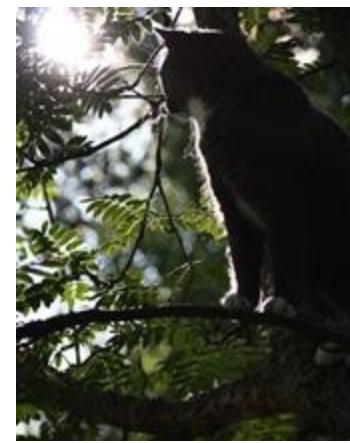
[This image](#) is [CC0 1.0](#) public domain



[This image](#) is [CC0 1.0](#) public domain



[This image](#) is [CC0 1.0](#) public domain



[This image](#) is [CC0 1.0](#) public domain

# Challenges: Background Clutter



[This image](#) is [CC0 1.0](#) public domain



[This image](#) is [CC0 1.0](#) public domain

# Challenges: Occlusion



[This image](#) is [CC0 1.0](#) public domain

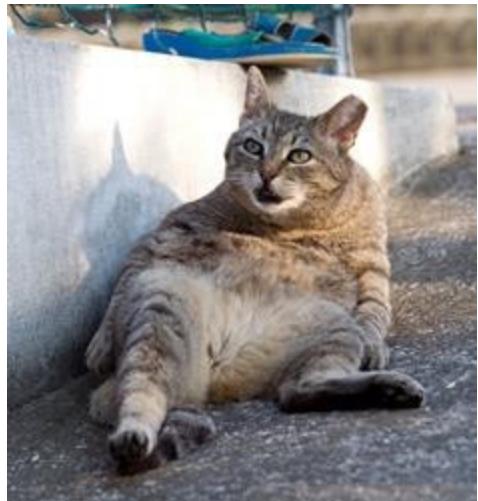


[This image](#) is [CC0 1.0](#) public domain



[This image](#) by [jonsson](#) is licensed  
under [CC-BY 2.0](#)

# Challenges: Deformation



[This image](#) by [Umberto Salvagnin](#) is  
licensed under [CC-BY 2.0](#)



[This image](#) by [Umberto Salvagnin](#) is  
licensed under [CC-BY 2.0](#)



[This image](#) by [sare bear](#) is  
licensed under [CC-BY 2.0](#)



[This image](#) by [Tom Thai](#) is licensed  
under [CC-BY 2.0](#)

# Challenges: Intraclass variation



[This image](#) is [CC0 1.0](#) public domain

# Challenges: Context

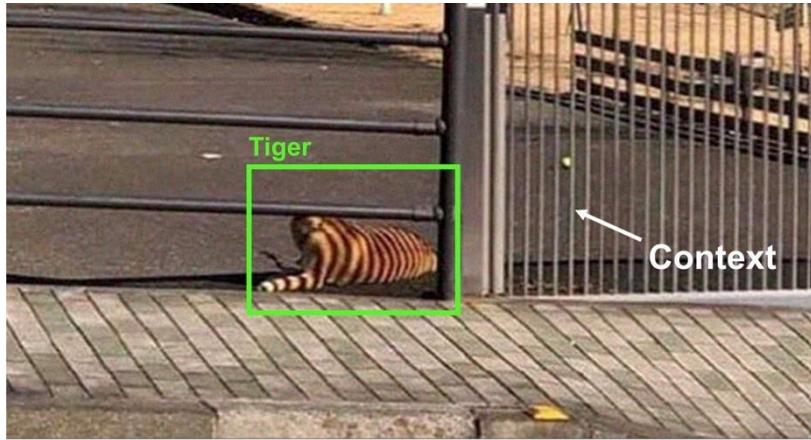
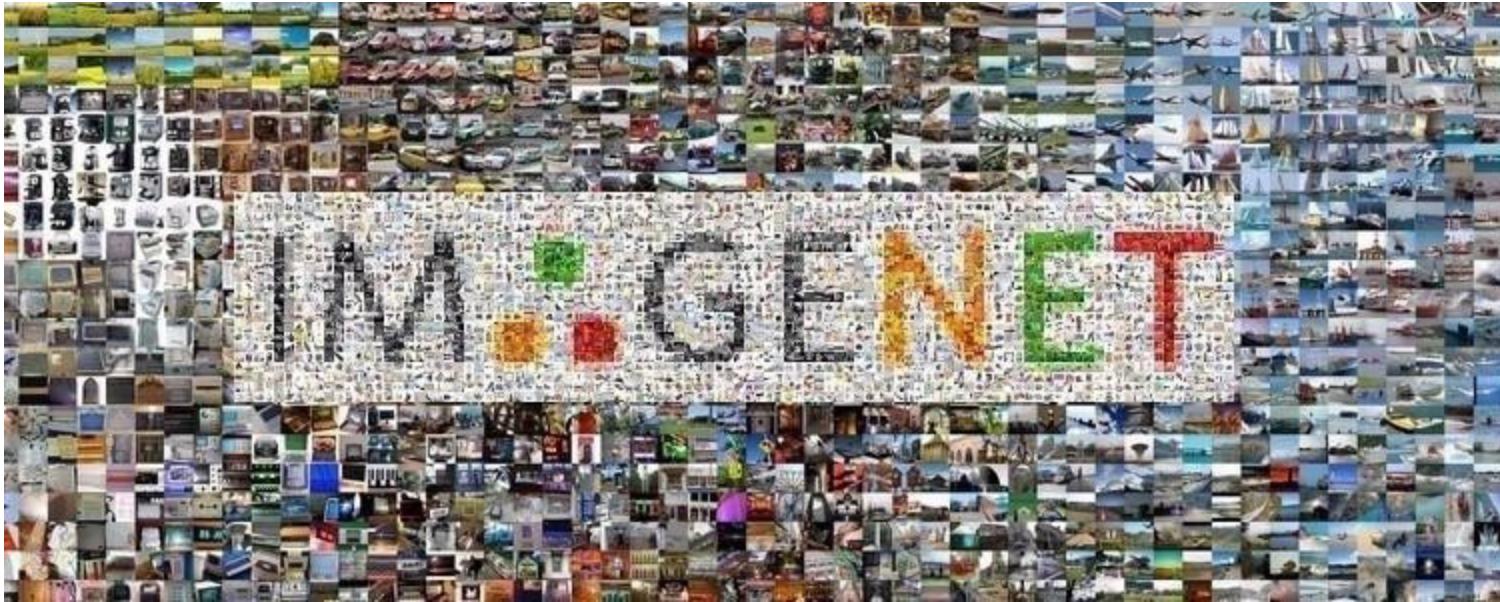


Image source: [https://www.linkedin.com/posts/ralph-aboujaoude-diaz-40838313\\_technology-artificialintelligence-computervision-activity-6912446088364875776-h-lq?utm\\_source=linkedin\\_share&utm\\_medium=member\\_desktop\\_web](https://www.linkedin.com/posts/ralph-aboujaoude-diaz-40838313_technology-artificialintelligence-computervision-activity-6912446088364875776-h-lq?utm_source=linkedin_share&utm_medium=member_desktop_web)

# Modern computer vision algorithms



[This image](#) is [CC0 1.0](#) public domain

# An image classifier

```
def classify_image(image):  
    # Some magic here?  
    return class_label
```

Unlike e.g. sorting a list of numbers,  
no obvious way to hard-code the algorithm for  
recognizing a cat, or other classes.

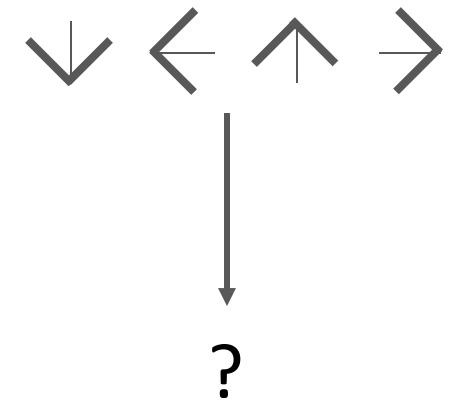
# Attempts have been made



Find edges



Find corners



# Machine Learning: Data-Driven Approach

1. Collect a dataset of images and labels
2. Use Machine Learning algorithms to train a classifier
3. Evaluate the classifier on new images

Example training set

```
def train(images, labels):  
    # Machine learning!  
    return model
```

---

```
def predict(model, test_images):  
    # Use model to predict labels  
    return test_labels
```



# Nearest Neighbor Classifier

# First classifier: Nearest Neighbor

```
def train(images, labels):  
    # Machine learning!  
    return model
```

→ Memorize all data  
and labels

```
def predict(model, test_images):  
    # Use model to predict labels  
    return test_labels
```

→ Predict the label of  
the most similar  
training image

# First classifier: Nearest Neighbor



deer



bird



plane



cat



car

Training data with labels



query data

Distance Metric



,



$\rightarrow \mathbb{R}$

# Distance Metric to compare images

L1 distance:

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

test image				training image				pixel-wise absolute value differences			
56	32	10	18	10	20	24	17	46	12	14	1
90	23	128	133	8	10	89	100	82	13	39	33
24	26	178	200	12	16	178	170	12	10	0	30
2	0	255	220	4	32	233	112	2	32	22	108

-

=

add → 456

```
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred
```

## Nearest Neighbor classifier

```
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred
```

## Nearest Neighbor classifier

Memorize training data

```
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred
```

## Nearest Neighbor classifier

For each test image:  
Find closest train image  
Predict label of nearest image

```
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred
```

## Nearest Neighbor classifier

Q: With N examples, how fast are training and prediction?

Ans: Train O(1),  
predict O(N)

This is bad: we want classifiers that are fast at prediction; slow for training is ok

```
import numpy as np

class NearestNeighbor:
    def __init__(self):
        pass

    def train(self, X, y):
        """ X is N x D where each row is an example. Y is 1-dimension of size N """
        # the nearest neighbor classifier simply remembers all the training data
        self.Xtr = X
        self.ytr = y

    def predict(self, X):
        """ X is N x D where each row is an example we wish to predict label for """
        num_test = X.shape[0]
        # lets make sure that the output type matches the input type
        Ypred = np.zeros(num_test, dtype = self.ytr.dtype)

        # loop over all test rows
        for i in xrange(num_test):
            # find the nearest training image to the i'th test image
            # using the L1 distance (sum of absolute value differences)
            distances = np.sum(np.abs(self.Xtr - X[i,:]), axis = 1)
            min_index = np.argmin(distances) # get the index with smallest distance
            Ypred[i] = self.ytr[min_index] # predict the label of the nearest example

        return Ypred
```

## Nearest Neighbor classifier

Many methods exist for fast / approximate nearest neighbor (beyond the scope of  $231N!$ )

A good implementation:

<https://github.com/facebookresearch/faiss>

Johnson et al, “Billion-scale similarity search with GPUs”, arXiv 2017

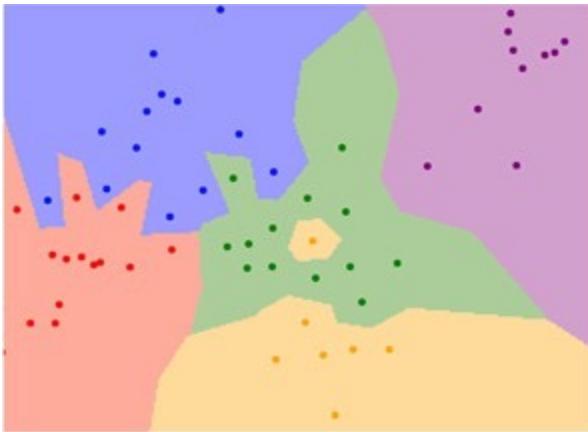
# What does this look like?



1-nearest neighbor

# K-Nearest Neighbors

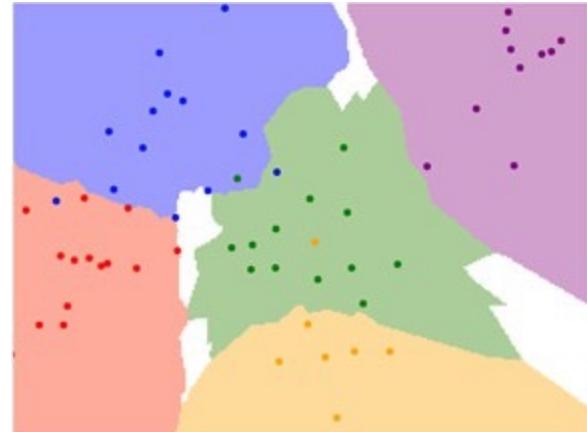
Instead of copying label from nearest neighbor,  
take majority vote from K closest points



$K = 1$



$K = 3$

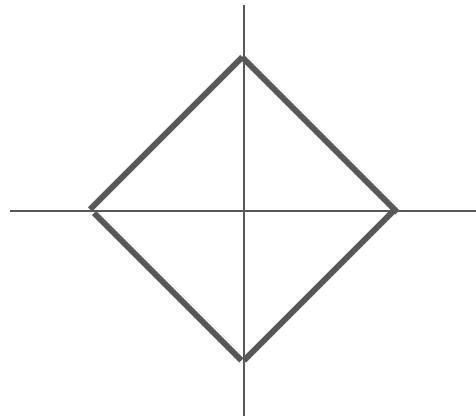


$K = 5$

# K-Nearest Neighbors: Distance Metric

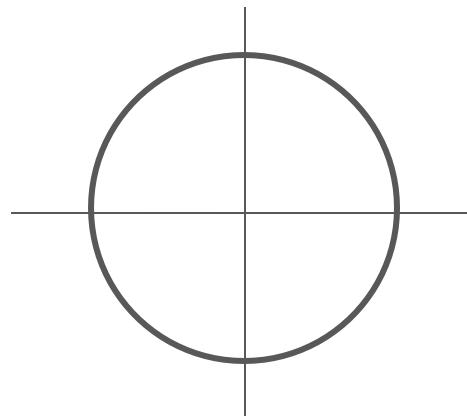
L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



L2 (Euclidean) distance

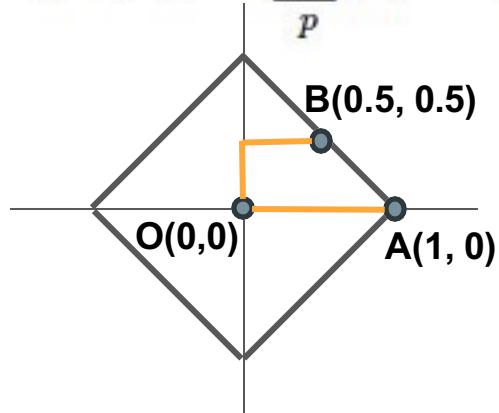
$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



# K-Nearest Neighbors: Distance Metric - Example

**L1 Distance:** Measures distance by moving along grid lines (like walking in a city with square blocks).

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



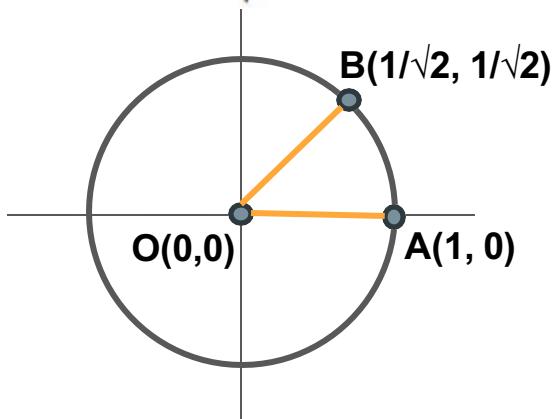
$$d_1(O, A) = |0-1| + |0-0| = 1$$

$$d_1(O, B) = |0-0.5| + |0-0.5| = 0.5 + 0.5 = 1$$

$$d_1(O, A) = d_1(O, B) = 1$$

**L2 Distance:** Measures the straight-line distance (as the crow flies).

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



$$d_2(O, A) = \sqrt{(0-1)^2 + (0-0)^2} = \sqrt{1^2} = 1$$

$$d_2(O, B) = \sqrt{(0-1/\sqrt{2})^2 + (0-1/\sqrt{2})^2} = \sqrt{1/2+1/2} = \sqrt{1} = 1$$

$$d_2(O, A) = d_2(O, B) = 1$$

# K-Nearest Neighbors: Distance Metric

L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



$K = 1$

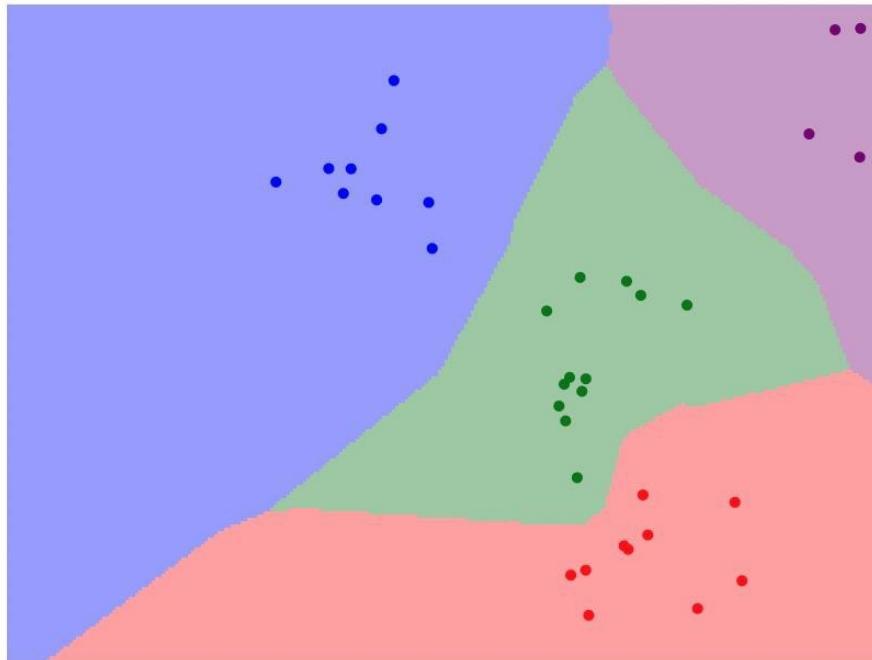
L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



$K = 1$

# K-Nearest Neighbors: try it yourself!



<http://vision.stanford.edu/teaching/cs231n-demos/knn/>

# Hyperparameters

What is the best value of k to use?

What is the best distance to use?

These are hyperparameters: choices about the algorithms themselves.

Very problem/dataset-dependent.

Must try them all out and see what works best.

# Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the training data

train

# Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the training data

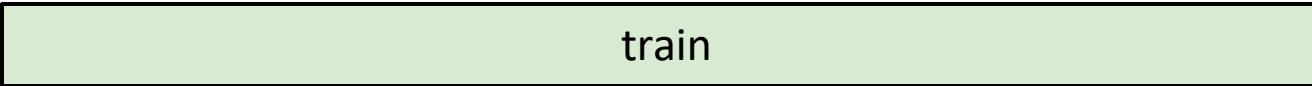
BAD:  $K = 1$  always works perfectly on training data

train

# Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the training data

BAD:  $K = 1$  always works perfectly on training data



train

Idea #2: choose hyperparameters that work best on test data



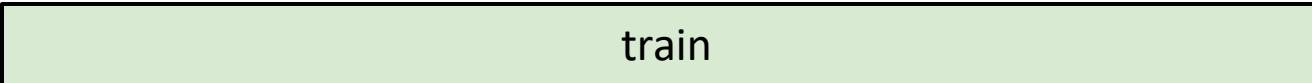
train

test

# Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the training data

BAD:  $K = 1$  always works perfectly on training data



train

Idea #2: choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data



train

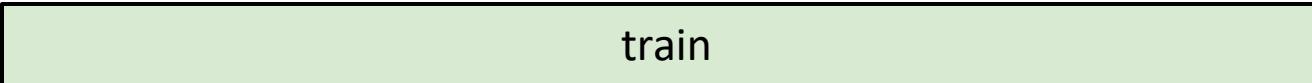
test

Never do this!

# Setting Hyperparameters

Idea #1: Choose hyperparameters that work best on the training data

BAD: K = 1 always works perfectly on training data



train

Idea #2: choose hyperparameters that work best on test data

BAD: No idea how algorithm will perform on new data



train

test

Idea #3: Split data into train, val; choose hyperparameters on val and evaluate on test

Better!



train

validation

test

# Setting Hyperparameters

train

Idea #4: Cross-Validation: Split data into folds, try each fold as validation and average the results

fold 1	fold 2	fold 3	fold 4	fold 5
fold 1	fold 2	fold 3	fold 4	fold 5
fold 1	fold 2	fold 3	fold 4	fold 5
fold 1	fold 2	fold 3	fold 4	fold 5
fold 1	fold 2	fold 3	fold 4	fold 5

test

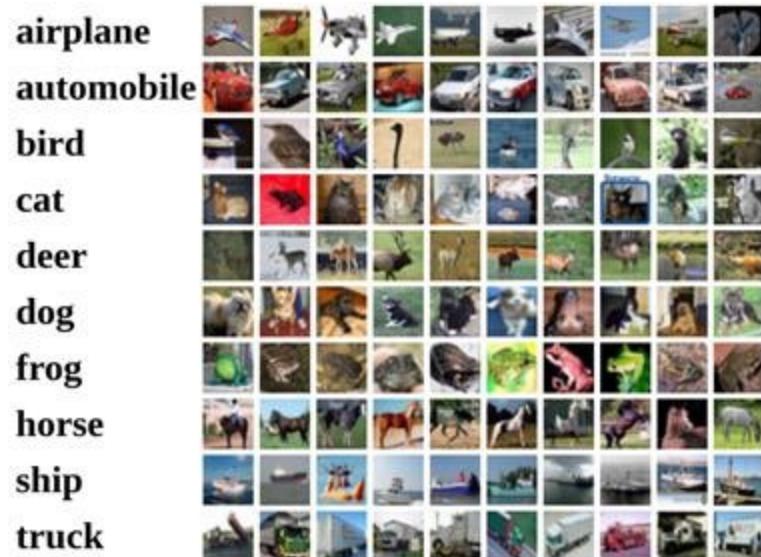
Useful for small datasets, but not used too frequently in deep learning

# Example Dataset: CIFAR10

10 classes

50,000 training images

10,000 testing images



# Example Dataset: CIFAR10

10 classes

50,000 training images

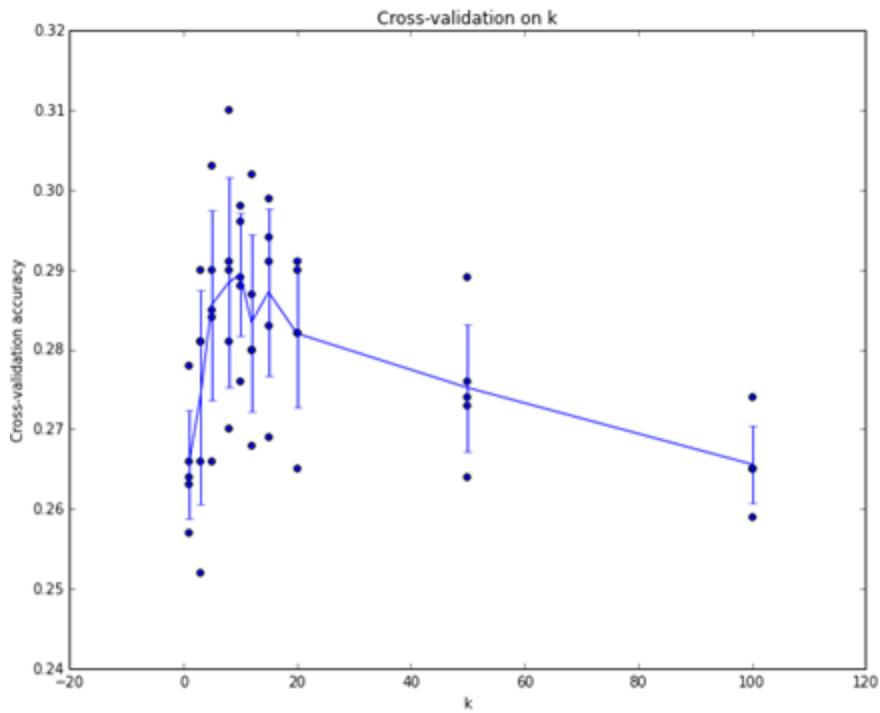
10,000 testing images



Test images and nearest neighbors



# Setting Hyperparameters



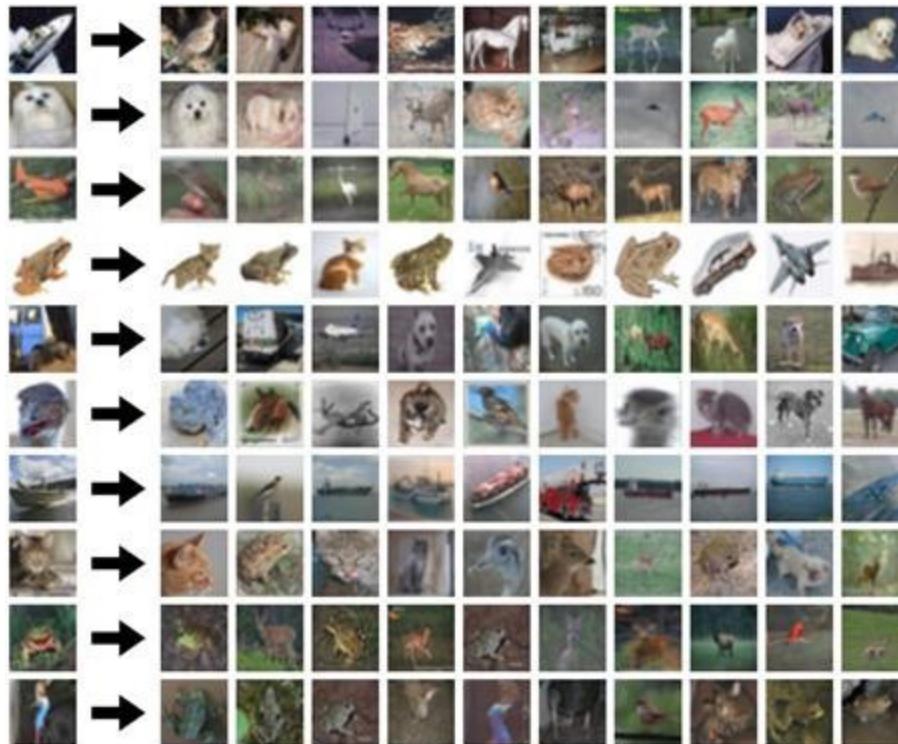
Example of  
5-fold cross-validation  
for the value of k.

Each point: single  
outcome.

The line goes  
through the mean, bars  
indicated standard  
deviation

(Seems that  $k \approx 7$  works best  
for this data)

# What does this look like?



# What does this look like?



# k-Nearest Neighbor with pixel distance never used.

- Distance metrics on pixels are not informative

[Original image is CC0  
public domain](#)



(All three images on the right have the same pixel distances to the one on the left)

# K-Nearest Neighbors: Summary

In image classification we start with a training set of images and labels, and must predict labels on the test set

The K-Nearest Neighbors classifier predicts labels based on the **K nearest training examples**

**Distance metric and K are hyperparameters**

Choose hyperparameters using the validation set

Only run on the test set once at the very end!

# Linear Classifier

# Parametric Approach

Image



Array of 32x32x3 numbers  
(3072 numbers total)

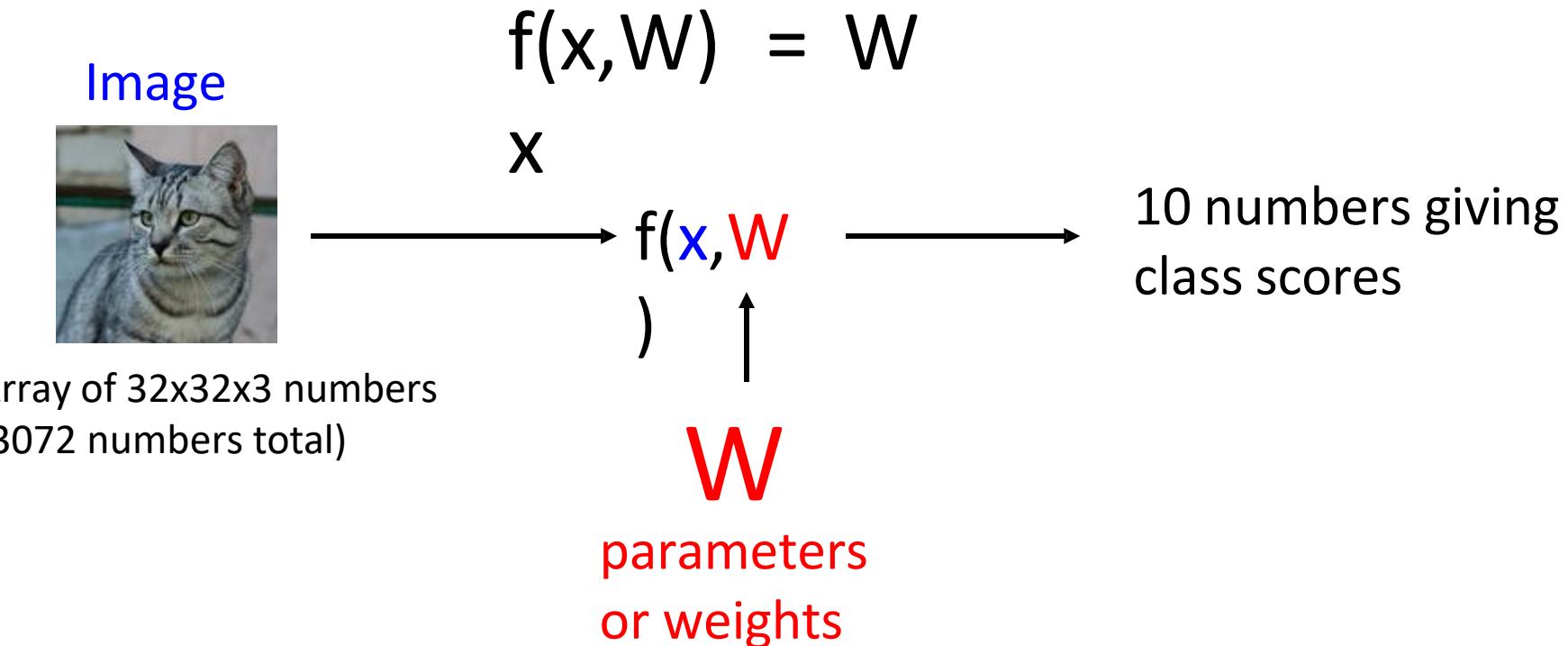


)  
↑

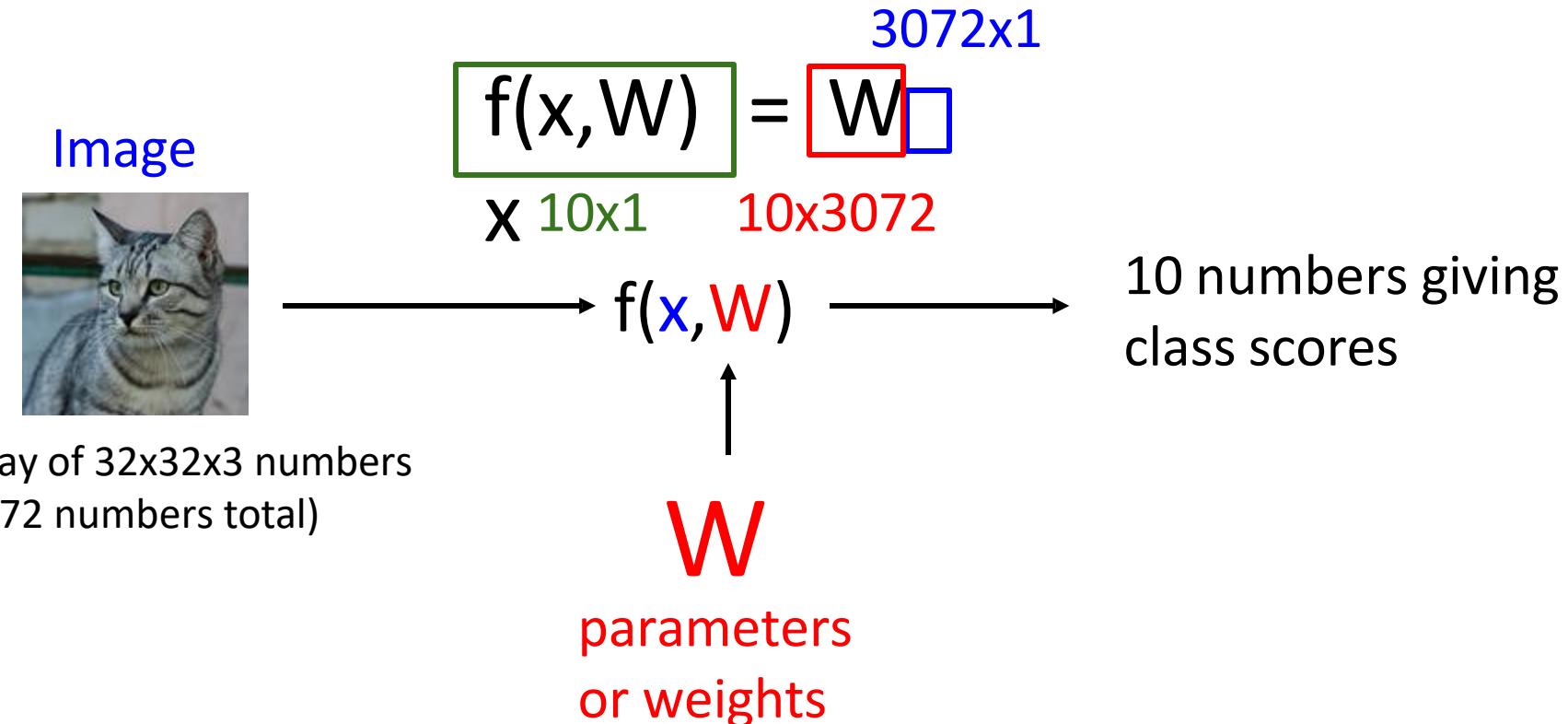
W  
parameters  
or weights

10 numbers giving  
class scores

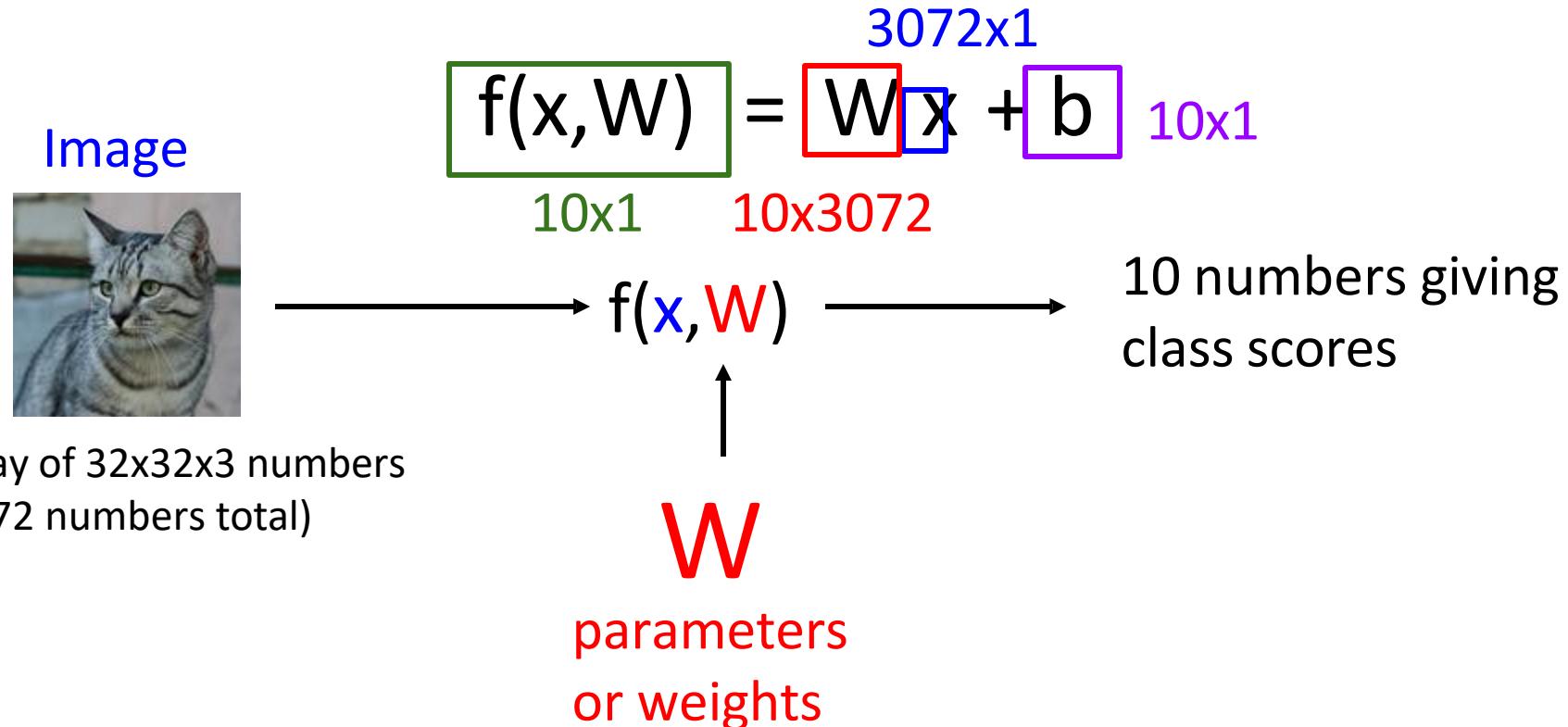
# Parametric Approach: Linear Classifier



# Parametric Approach: Linear Classifier



# Parametric Approach: Linear Classifier

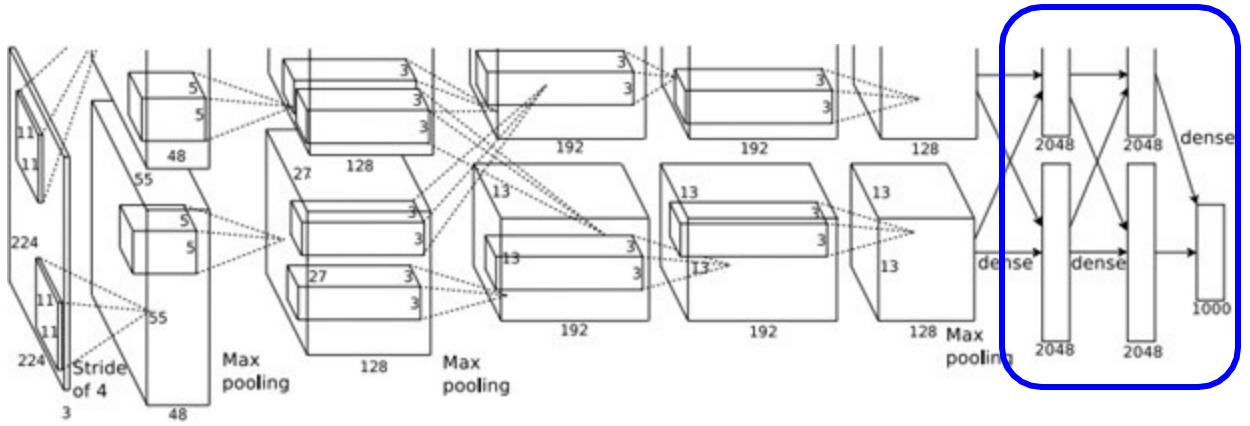


# Neural Network

Linear  
classifiers

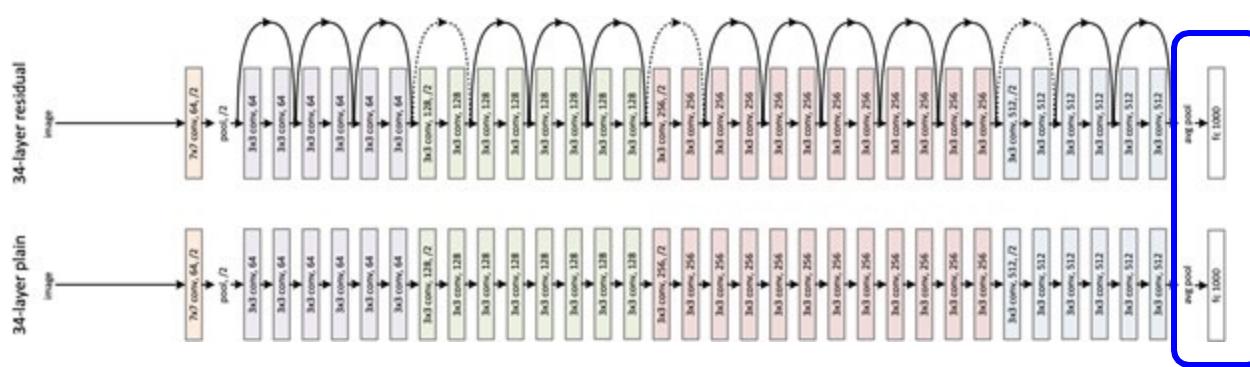


[This image](#) is [CC0 1.0](#) public domain



[Krizhevsky et al. 2012]

## Linear layers



[He et al. 2015]

# Recall CIFAR10

**airplane**



**automobile**



**bird**



**cat**



**deer**



**dog**



**frog**



**horse**



**ship**



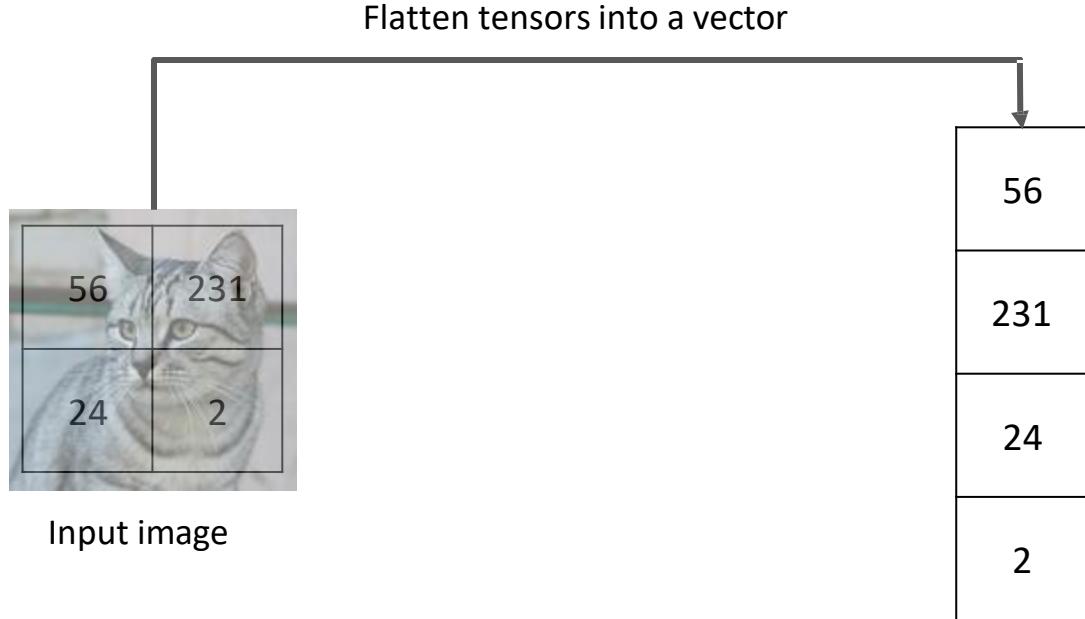
**truck**



50,000 training images  
each image is 32x32x3

10,000 test images.

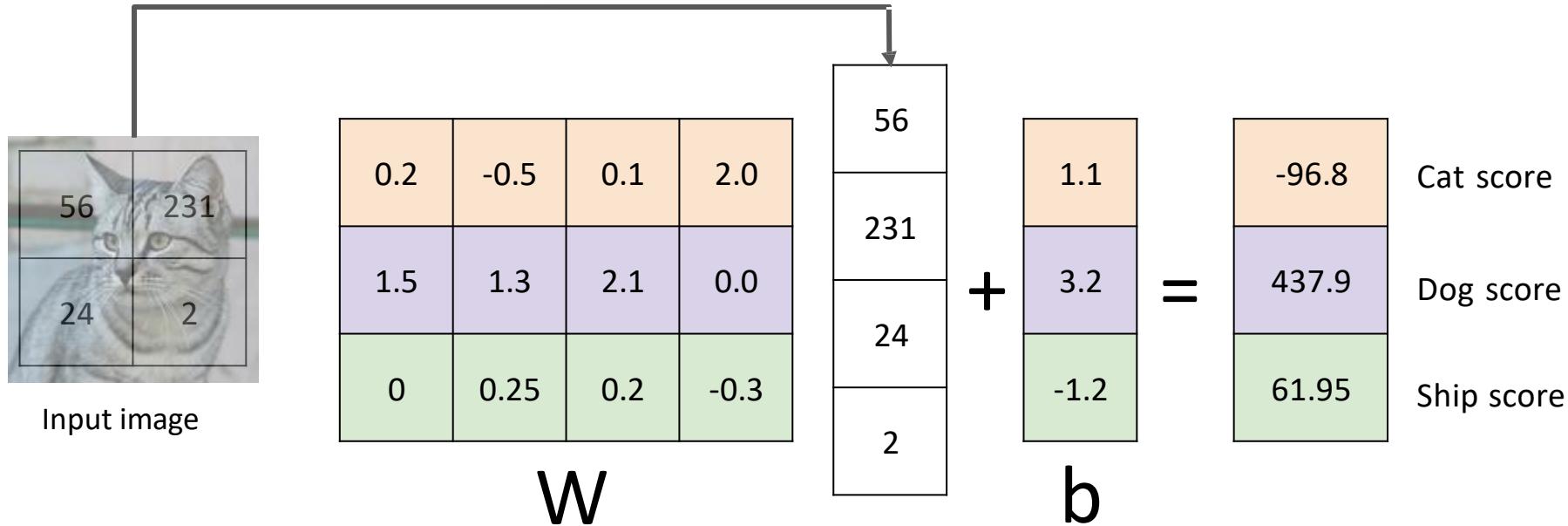
Example with an image with 4 pixels, and 3 classes (**cat/dog/ship**)



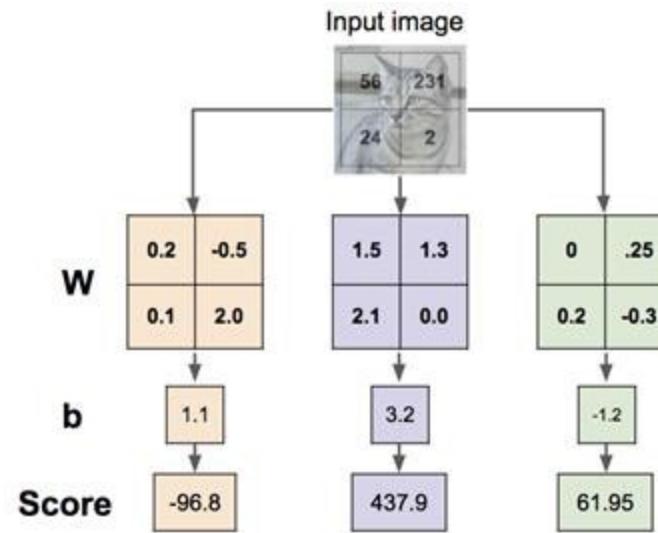
Example with an image with 4 pixels, and 3 classes (**cat/dog/ship**)

## Algebraic Viewpoint

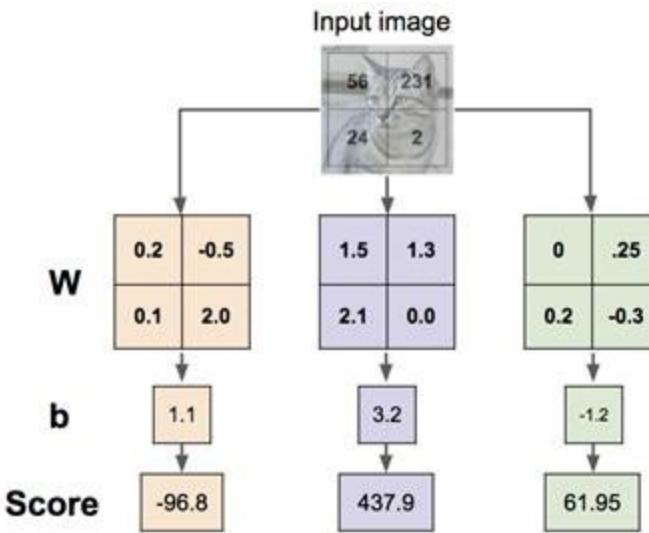
Flatten tensors into a vector



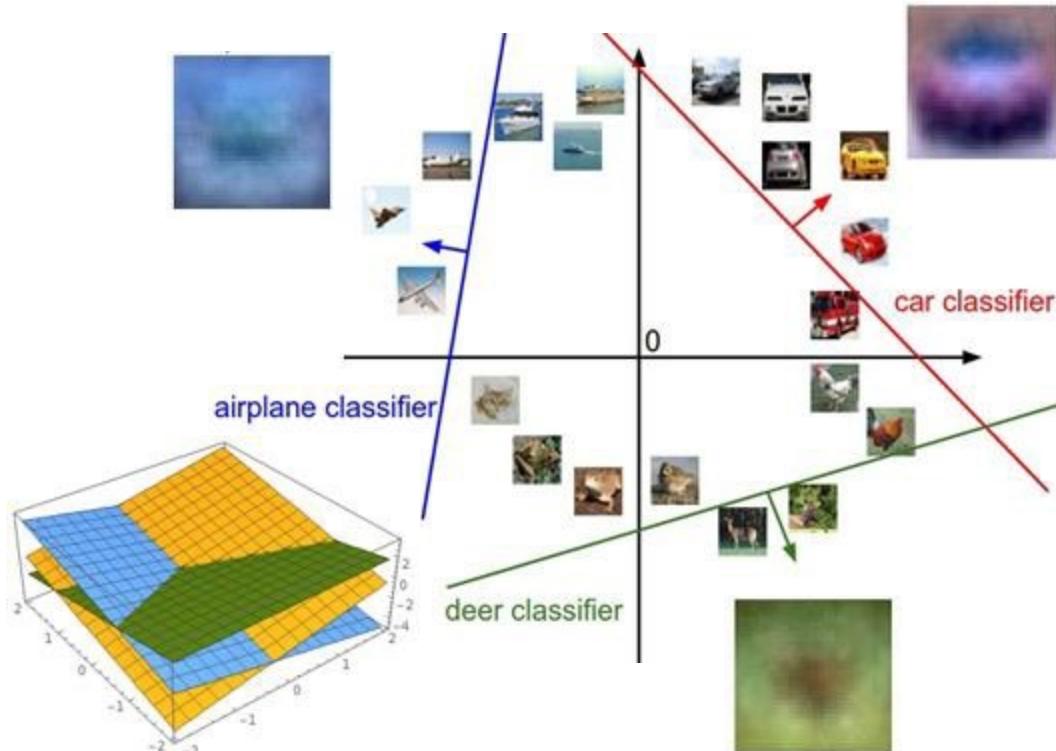
# Interpreting a Linear Classifier



# Interpreting a Linear Classifier: Visual Viewpoint



# Interpreting a Linear Classifier: Geometric Viewpoint



$$f(x, W) = Wx + b$$



Array of 32x32x3 numbers  
(3072 numbers total)

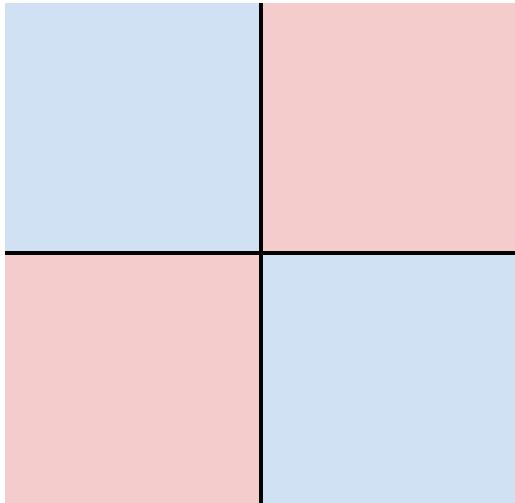
# Hard cases for a linear classifier

Class 1:

First and third quadrants

Class 2:

Second and fourth quadrants

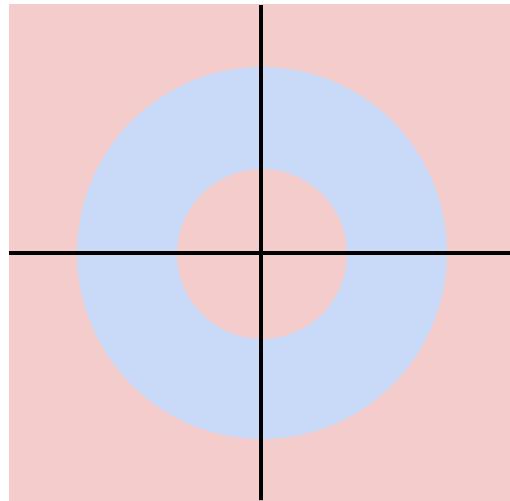


Class 1:

$1 \leq L_2 \text{ norm} \leq 2$

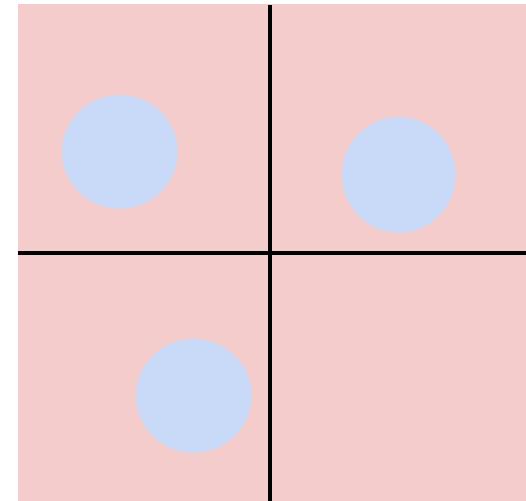
Class 2:

Everything else



Class 1: Three modes

Class 2:  
Everything else



# Linear Classifier – Choose a good W



airplane	-3.45	-0.51	3.42
automobile	-8.87	<b>6.04</b>	4.64
bird	0.09	5.31	2.65
cat	<b>2.9</b>	-4.22	5.1
deer	4.48	-4.19	2.64
dog	8.02	3.58	5.55
frog	3.78	4.49	<b>-4.34</b>
horse	1.06	-4.37	-1.5
ship	-0.36	-2.09	-4.79
truck	-0.72	-2.93	6.14

1. Define a loss function that quantifies our unhappiness with the scores across the training data.
2. Come up with a way of efficiently finding the parameters that minimize the loss function. (optimization)

Suppose: 3 training examples, 3 classes.

With some  $W$  the scores  $f(x, W) = Wx$



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

Suppose: 3 training examples, 3 classes.

With some  $W$  the scores  $f(x, W) = Wx$

A loss function tells how good our current classifier is



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

Suppose: 3 training examples, 3 classes.

With some  $W$  the scores  $f(x, W) = Wx$



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

A loss function tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^N$$

Where  $x_i$  is image and  
e  $y_i$  is (integer) label

Suppose: 3 training examples, 3 classes.

With some  $W$  the scores  $f(x, W) = Wx$



cat	3.2	1.3	2.2
car	5.1	4.9	2.5
frog	-1.7	2.0	-3.1

A loss function tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^N$$

Where  $x_i$  is image and  
e  $y_i$  is (integer) label

Loss over the dataset is a average of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

# Softmax classifier

# Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as probabilities



cat	3.2
car	5.1
frog	-1.7

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax  
Function

cat            3.2

car            5.1

frog          -1.7

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax  
Function

Probabilities  
must be  $\geq 0$

cat	3.2
car	5.1
frog	-1.7

$$\begin{array}{r} 24.5 \\ \longrightarrow \\ 164.0 \\ 0.18 \end{array}$$

unnormalized  
probabilities

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

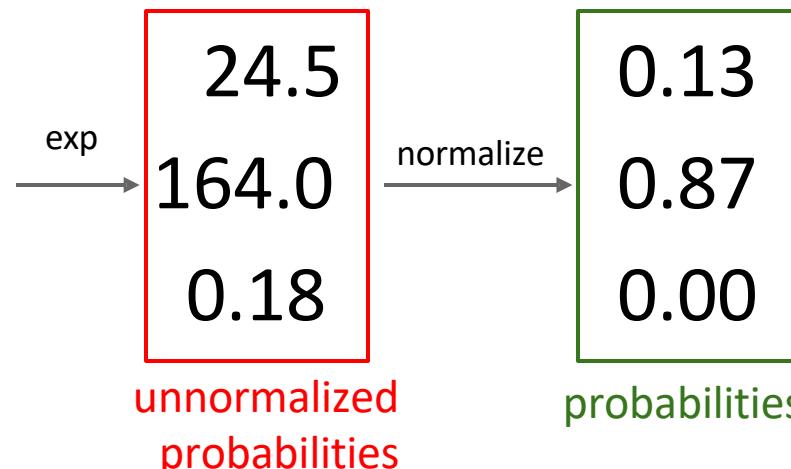
$$s = f(x_i; W)$$

Probabilities  
must be  $\geq 0$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax  
Function

cat            3.2  
car            5.1  
frog          -1.7



# Softmax Classifier (Multinomial Logistic Regression)

Want to interpret raw classifier scores as probabilities



$$s = f(x_i; W)$$

Probabilities  
must be  $\geq 0$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax  
Function

cat  
car  
frog

3.2
5.1
-1.7

Unnormalized log-  
probabilities / logits

exp

24.5
164.0
0.18

unnormalized  
probabilities

normalize

0.13
0.87
0.00

probabilities

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

cat  
car  
frog

3.2
5.1
-1.7

Unnormalized log-probabilities / logits

exp

24.5
164.0
0.18

unnormalized probabilities

normalize

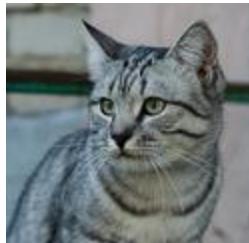
0.13
0.87
0.00

probabilities

$$L_i = -\log P(Y = y_i|X = x_i)$$

$$\rightarrow L_i = -\log(0.13) \\ = 2.04$$

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

cat  
car  
frog

3.2
5.1
-1.7

Unnormalized log-probabilities / logits

exp

24.5
164.0
0.18

unnormalized probabilities

normalize

0.13
0.87
0.00

probabilities

$$L_i = -\log P(Y = y_i|X = x_i)$$

$$\rightarrow L_i = -\log(0.13) \\ = 2.04$$

Maximum Likelihood Estimation  
Choose weights to maximize the likelihood of the observed data  
(See CS 229 for details)

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

cat  
car  
frog

3.2
5.1
-1.7

Unnormalized log-probabilities / logits

24.5
164.0
0.18

unnormalized probabilities

Probabilities must be  $\geq 0$

Probabilities must sum to 1

0.13
0.87
0.00

probabilities

$$L_i = -\log P(Y = y_i|X = x_i)$$

compare

1.00
0.00
0.00

Correct probs

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

cat  
car  
frog

3.2
5.1
-1.7

Unnormalized log-probabilities / logits

Probabilities must be  $\geq 0$

24.5
164.0
0.18

unnormalized probabilities

exp

normalize

Probabilities must sum to 1

0.13
0.87
0.00

probabilities

$$L_i = -\log P(Y = y_i|X = x_i)$$

$$\begin{aligned} &\text{compare} \\ &\text{Kullback–Leibler divergence} \\ &D_{KL}(P||Q) = \sum_y P(y) \log \frac{P(y)}{Q(y)} \end{aligned}$$

1.00

0.00

0.00

Correct probs

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax Function

cat  
car  
frog

3.2
5.1
-1.7

Unnormalized log-probabilities / logits

24.5
164.0
0.18

unnormalized probabilities

Probabilities must sum to 1

0.13
0.87
0.00

probabilities

$$L_i = -\log P(Y = y_i|X = x_i)$$

compare

1.00

0.00

0.00

Cross Entropy

$$H(P, Q) = H(p) + D_{KL}(P||Q)$$

Correct probs

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax  
Function

Maximize probability of correct class

Putting it all together:

cat	3.2
car	5.1
frog	-1.7

$$L_i = -\log P(Y = y_i|X = x_i) \quad L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax  
Function

Maximize probability of correct class

Putting it all together:

$$L_i = -\log P(Y = y_i|X = x_i)$$

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

cat	3.2
car	5.1
frog	-1.7

Q1: What is the min/max possible softmax loss  $L_i$ ?

Q2: At initialization all  $s_j$  will be approximately equal; what is the softmax loss  $L_i$ , assuming  $C$  classes?

# Softmax Classifier (Multinomial Logistic Regression)



Want to interpret raw classifier scores as probabilities

$$s = f(x_i; W)$$

$$P(Y = k|X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

Softmax  
Function

cat	3.2
car	5.1
frog	-1.7

Maximize probability of correct class

$$L_i = -\log P(Y = y_i|X = x_i)$$

Putting it all together:

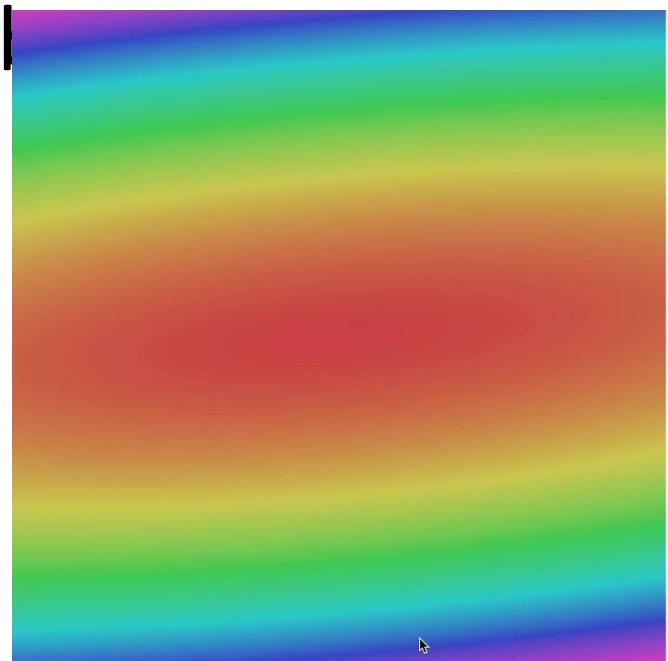
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Q2: At initialization all  $s$  will be approximately equal; what is the loss?  
A:  $-\log(1/C) = \log(C)$ ,  
If  $C = 10$ , then  $L_i = \log(10) \approx 2.3$

# Coming up:

- Regularization
- Optimization

$$f(x, W) = Wx +$$



# Lecture 3: Regularization and Optimization

# Image Classification: A core task in Computer Vision



(assume given a set of labels)  
{dog, cat, truck, plane, ...}

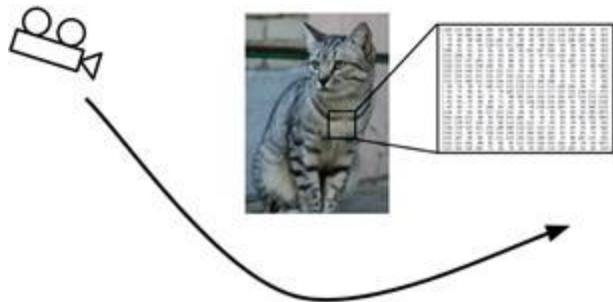


cat  
dog  
bird  
deer  
truck

This image by [Nikita](#) is  
licensed under [CC-BY 2.0](#)

# Recall from last time: Challenges of recognition

Viewpoint



Illumination



[This image](#) is CC0 1.0 public domain

Deformation



[This image](#) by Umberto Salvagnin  
is licensed under CC-BY 2.0

Occlusion



[This image](#) by ionsson is licensed  
under CC-BY 2.0

Clutter



[This image](#) is CC0 1.0 public domain

Intraclass Variation



[This image](#) is CC0 1.0 public domain

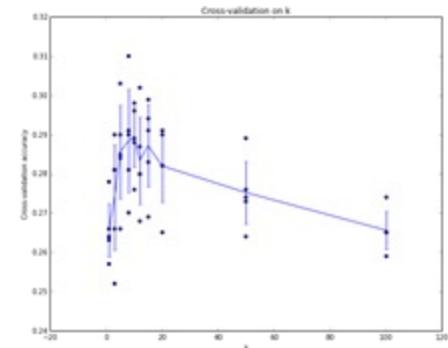
# Recall from last time: data-driven approach, kNN



1-NN classifier



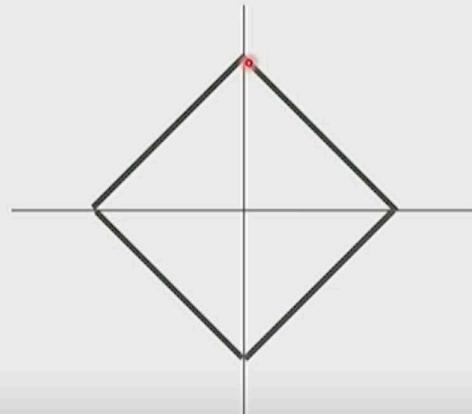
5-NN classifier



# Recall from last time: Distance Metric

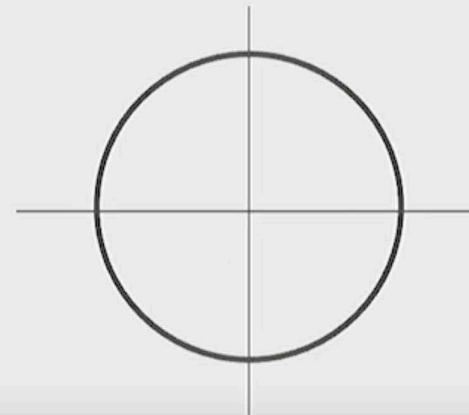
L1 (Manhattan) distance

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



L2 (Euclidean) distance

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$

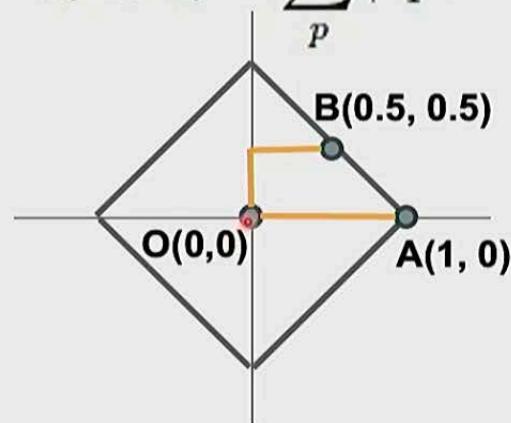


Stanford

# Recall from last time: Distance Metric - Example

**L1 Distance:** Measures distance by moving along grid lines (like walking in a city with square blocks).

$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$



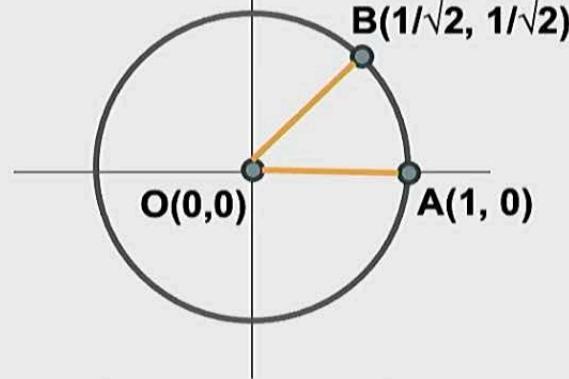
$$d_1(O, A) = |0-1| + |0-0| = 1$$

$$d_1(O, B) = |0-0.5| + |0-0.5| = 0.5 + 0.5 = 1$$

$$d_1(O, A) = d_1(O, B) = 1$$

**L2 Distance:** Measures the straight-line distance (as the crow flies).

$$d_2(I_1, I_2) = \sqrt{\sum_p (I_1^p - I_2^p)^2}$$



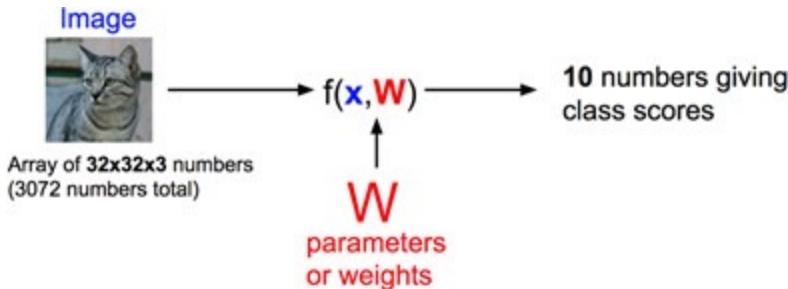
$$d_2(O, A) = \sqrt{(0-1)^2 + (0-0)^2} = \sqrt{1^2} = 1$$

$$d_2(O, B) = \sqrt{(0-1/\sqrt{2})^2 + (0-1/\sqrt{2})^2} = \sqrt{1/2+1/2} = \sqrt{1} = 1$$

$$d_2(O, A) = d_2(O, B) = 1$$

Stanford

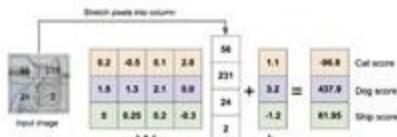
# Recall from last time: Linear Classifier



$$f(x, W) = Wx + b$$

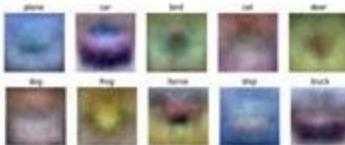
## Algebraic Viewpoint

$$f(x, W) = Wx$$



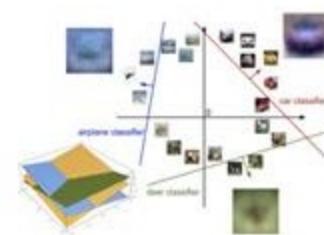
## Visual Viewpoint

One template  
per class



Geometric Viewpoint

Hyperplanes  
cutting up space



### Class 1:

## Class 2: Everything else

## Class 1: Three modes

### **Class 2:** Everything else

Suppose: 3 training examples, 3 classes.

With some  $W$  the scores  $f(x, W) = Wx$  are:



cat	<b>3.2</b>	1.3	2.2
car	5.1	<b>4.9</b>	2.5
frog	-1.7	2.0	<b>-3.1</b>

A **loss function** tells how good our current classifier is

Given a dataset of examples

$$\{(x_i, y_i)\}_{i=1}^N$$

Where  $x_i$  is image and  
 $y_i$  is (integer) label

Loss over the dataset is a average of loss over examples:

$$L = \frac{1}{N} \sum_i L_i(f(x_i, W), y_i)$$

# Regularization -

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}}$$

**Data loss:** Model predictions  
should match training data

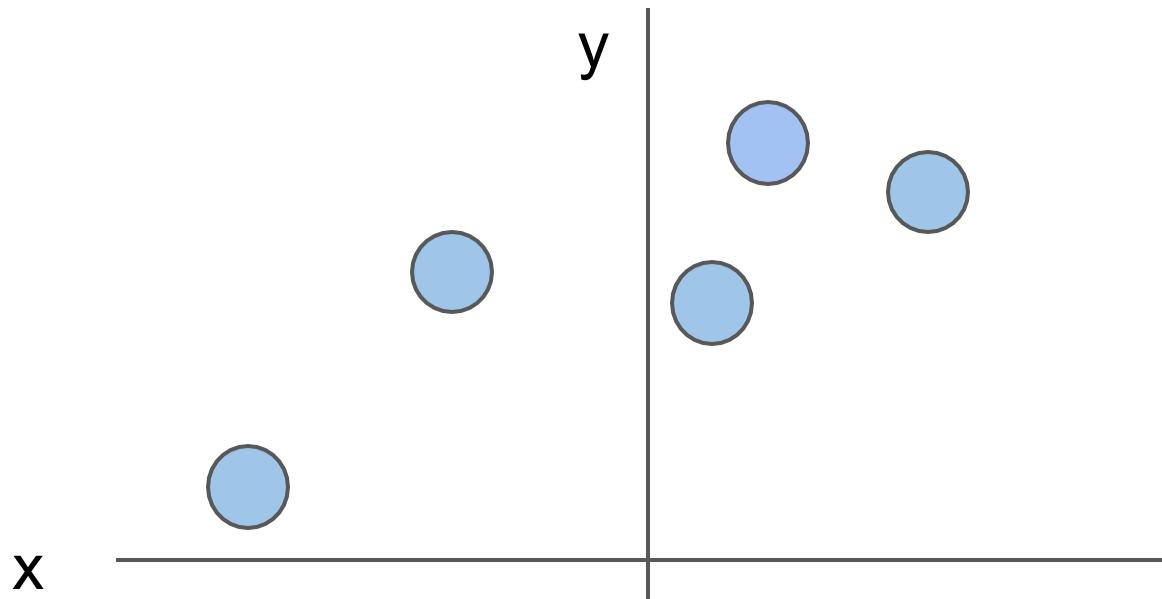
# Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \lambda R(W)$$

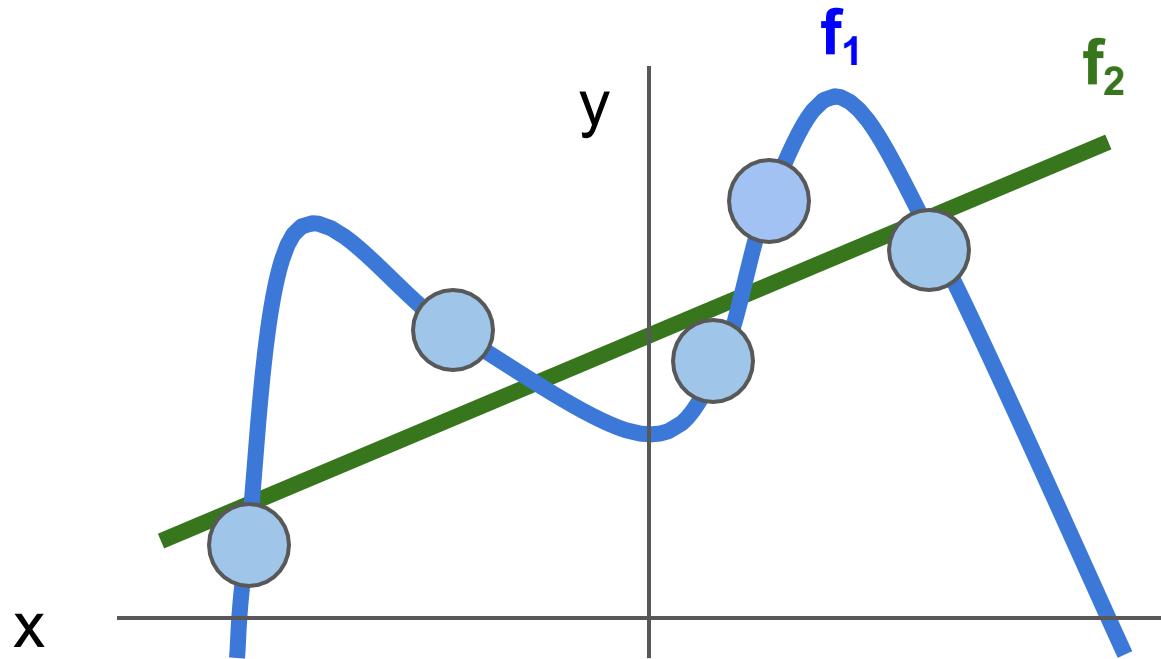

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too well* on training data

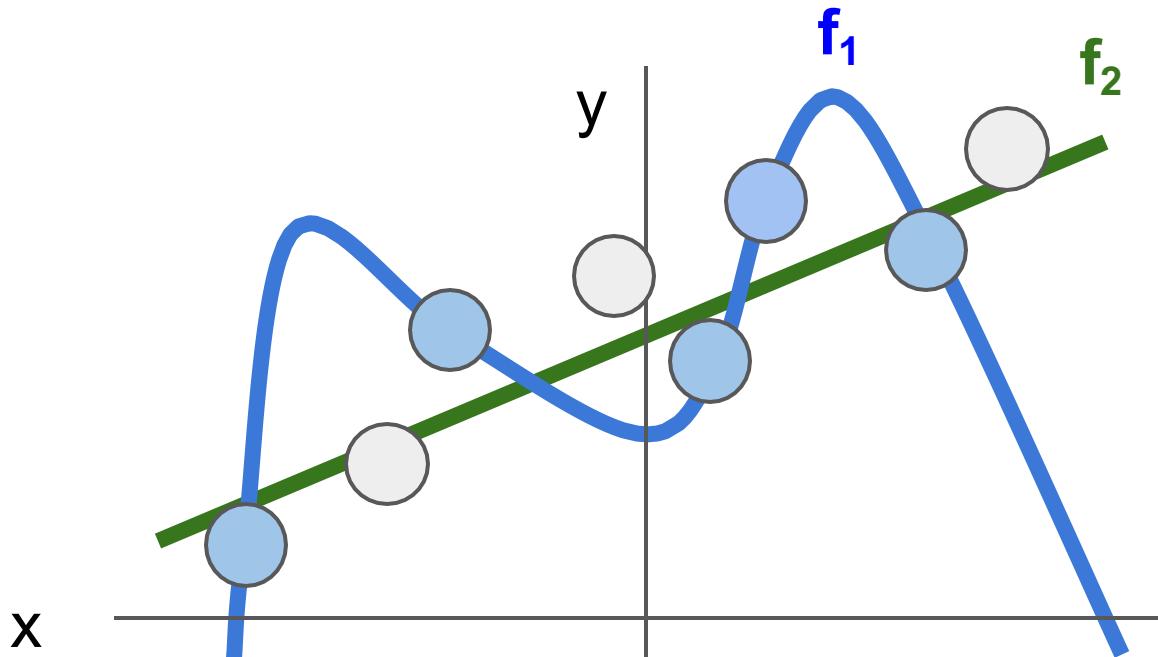
# Regularization intuition: toy example training data



# Regularization intuition: Prefer Simpler Models



# Regularization: Prefer Simpler Models



Regularization pushes against fitting the data  
too well so we don't fit noise in the data

# Regularization

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \lambda R(W)$$


**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too well* on training data

**Occam's Razor:** Among multiple competing hypotheses, the simplest is the best, William of Ockham 1285-1347

# Regularization

$\lambda$  = regularization strength  
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss: Model predictions should match training data}} + \lambda R(W)$$


**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too well* on training data

# Regularization

$\lambda$  = regularization strength  
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \lambda R(W)$$

**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too well* on training data

## Simple examples

L2 regularization:  $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization:  $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2):  $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

# Regularization

$\lambda$  = regularization strength  
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \lambda R(W)$$

**Data loss:** Model predictions should match training data



**Regularization:** Prevent the model from doing *too well* on training data

## Simple examples

L2 regularization:  $R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization:  $R(W) = \sum_k \sum_l |W_{k,l}|$

Elastic net (L1 + L2):  $R(W) = \sum_k \sum_l \beta W_{k,l}^2 + |W_{k,l}|$

## More complex:

Dropout

Batch normalization

Stochastic depth, fractional pooling, etc

# Regularization

$\lambda$  = regularization strength  
(hyperparameter)

$$L(W) = \underbrace{\frac{1}{N} \sum_{i=1}^N L_i(f(x_i, W), y_i)}_{\text{Data loss}} + \lambda R(W)$$


**Data loss:** Model predictions should match training data

**Regularization:** Prevent the model from doing *too well* on training data

Why regularize?

- Express preferences over weights
- Make the model *simple* so it works on test data
- Improve optimization by adding curvature

# Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will  
the L2 regularizer prefer?

$$w_1^T x = w_2^T x = 1$$

# Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will  
the L2 regularizer prefer?

L2 regularization likes to  
“spread out” the weights

$$w_1^T x = w_2^T x = 1$$

# Regularization: Expressing Preferences

$$x = [1, 1, 1, 1]$$

$$w_1 = [1, 0, 0, 0]$$

$$w_2 = [0.25, 0.25, 0.25, 0.25]$$

L2 Regularization

$$R(W) = \sum_k \sum_l W_{k,l}^2$$

Which of w1 or w2 will  
the L2 regularizer prefer?

L2 regularization likes to  
“spread out” the weights

$$w_1^T x = w_2^T x = 1$$

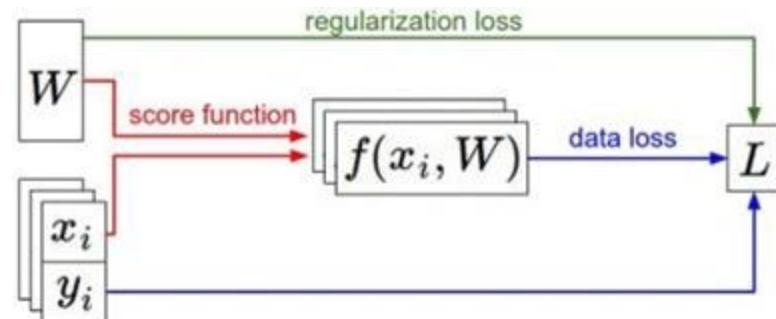
Which one would L1  
regularization prefer?

# Recap

- We have some dataset of  $(x, y)$
- We have a **score function**:  $s = f(x; W) = Wx$  e.g.
- We have a **loss function**:

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{sj}}\right) \text{ Softmax}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W) \text{ Full loss}$$



# Optimization



This image is [CC0 1.0](#) public domain



[Walking man image](#) is [CC0 1.0](#) public domain

# Strategy #1: A first very bad idea solution: Random search

```
# assume X_train is the data where each column is an example (e.g. 3073 x 50,000)
# assume Y_train are the labels (e.g. 1D array of 50,000)
# assume the function L evaluates the loss function

bestloss = float("inf") # Python assigns the highest possible float value
for num in xrange(1000):
    W = np.random.randn(10, 3073) * 0.0001 # generate random parameters
    loss = L(X_train, Y_train, W) # get the loss over the entire training set
    if loss < bestloss: # keep track of the best solution
        bestloss = loss
        bestW = W
    print 'in attempt %d the loss was %f, best %f' % (num, loss, bestloss)

# prints:
# in attempt 0 the loss was 9.401632, best 9.401632
# in attempt 1 the loss was 8.959668, best 8.959668
# in attempt 2 the loss was 9.044034, best 8.959668
# in attempt 3 the loss was 9.278948, best 8.959668
# in attempt 4 the loss was 8.857370, best 8.857370
# in attempt 5 the loss was 8.943151, best 8.857370
# in attempt 6 the loss was 8.605604, best 8.605604
# ... (truncated: continues for 1000 lines)
```

# Lets see how well this works on the test set...

```
# Assume X_test is [3073 x 10000], Y_test [10000 x 1]
scores = Wbest.dot(Xte_cols) # 10 x 10000, the class scores for all test examples
# find the index with max score in each column (the predicted class)
Yte_predict = np.argmax(scores, axis = 0)
# and calculate accuracy (fraction of predictions that are correct)
np.mean(Yte_predict == Yte)
# returns 0.1555
```

15.5% accuracy! not bad!  
(SOTA is ~99.7%)

## Strategy #2: Follow the slope



## Strategy #2: Follow the slope

In 1-dimension, the derivative of a function:

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

In multiple dimensions, the **gradient** is the vector of (partial derivatives) along each dimension

The slope in any direction is the **dot product** of the direction with the gradient  
The direction of steepest descent is the **negative gradient**

**current W:**

[0.34,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25347**

**gradient dW:**

[?,  
?,  
?,  
?,  
?,  
?,  
?,  
?,  
?,  
?,...]

current W:	W + h (first dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[0.34 + 0.0001, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[?, ?, ?, ?, ?, ?, ?, ?, ?,...]
<b>loss 1.25347</b>	<b>loss 1.25322</b>	

**current W:**

[0.34,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25347**

**W + h (first dim):**

[0.34 + 0.0001,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25322**

**gradient dW:**

**-2.5,**

?,

?,

$$\begin{aligned} & (1.25322 - 1.25347) / 0.0001 \\ & = -2.5 \end{aligned}$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

?,

?,...]

current W:	W + h (second dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[0.34, <b>-1.11 + 0.0001</b> , 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[-2.5, ?, ?, ?, ?, ?, ?, ?, ?, ?,...]
<b>loss 1.25347</b>	<b>loss 1.25353</b>	

**current W:**

[0.34,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25347**

**W + h (second dim):**

[0.34,  
-1.11 + **0.0001**,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25353**

**gradient dW:**

**[-2.5,**  
**0.6,**  
**?,**  
**?,**

$$\frac{(1.25353 - 1.25347)}{0.0001} = 0.6$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

**?,...]**

current W:	W + h (third dim):	gradient dW:
[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[0.34, -1.11, 0.78 + <b>0.0001</b> , 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]	[-2.5, 0.6, ?, ?, ?, ?, ?, ?, ?, ?,...]
<b>loss 1.25347</b>	<b>loss 1.25347</b>	

current W:

[0.34,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

loss 1.25347

W + h (third dim):

[0.34,  
-1.11,  
0.78 + 0.0001,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

loss 1.25347

gradient dW:

[-2.5,  
0.6,  
0,  
?,  
0]

$$\frac{(1.25347 - 1.25347)}{0.0001} = 0$$

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

?, ...]

**current W:**

[0.34,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25347**

**W + h (third dim):**

[0.34,  
-1.11,  
0.78 + **0.0001**,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25347**

**gradient dW:**

[-2.5,  
0.6,  
0,  
?,  
?]

### Numeric Gradient

- Slow! Need to loop over all dimensions
- Approximate

[,...]

This is silly. The loss is just a function of  $W$ :

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want  $\nabla_W L$

# This is silly. The loss is just a function of W:

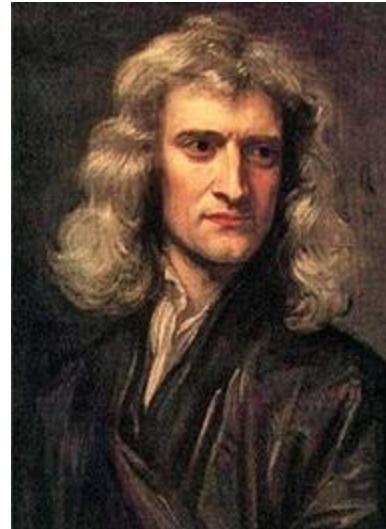
$$L = \frac{1}{N} \sum_{i=1}^N L_i + \sum_k W_k^2$$

$$L_i = -\log\left(\frac{e^{s_i}}{\sum_j e^{s_j}}\right)$$

$$s = f(x; W) = Wx$$

want  $\nabla_W L$

Use calculus to compute an analytic gradient



[This image](#) is in the public domain



[This image](#) is in the public domain

**current W:**

[0.34,  
-1.11,  
0.78,  
0.12,  
0.55,  
2.81,  
-3.1,  
-1.5,  
0.33,...]

**loss 1.25347**

**gradient dW:**

[-2.5,  
0.6,  
0,  
0.2,  
0.7,  
-0.5,  
1.1,  
1.3,  
-2.1,...]

dW = ...  
(some function  
data and W)



# In summary:

- Numerical gradient: approximate, slow, easy to write
- Analytic gradient: exact, fast, error-prone

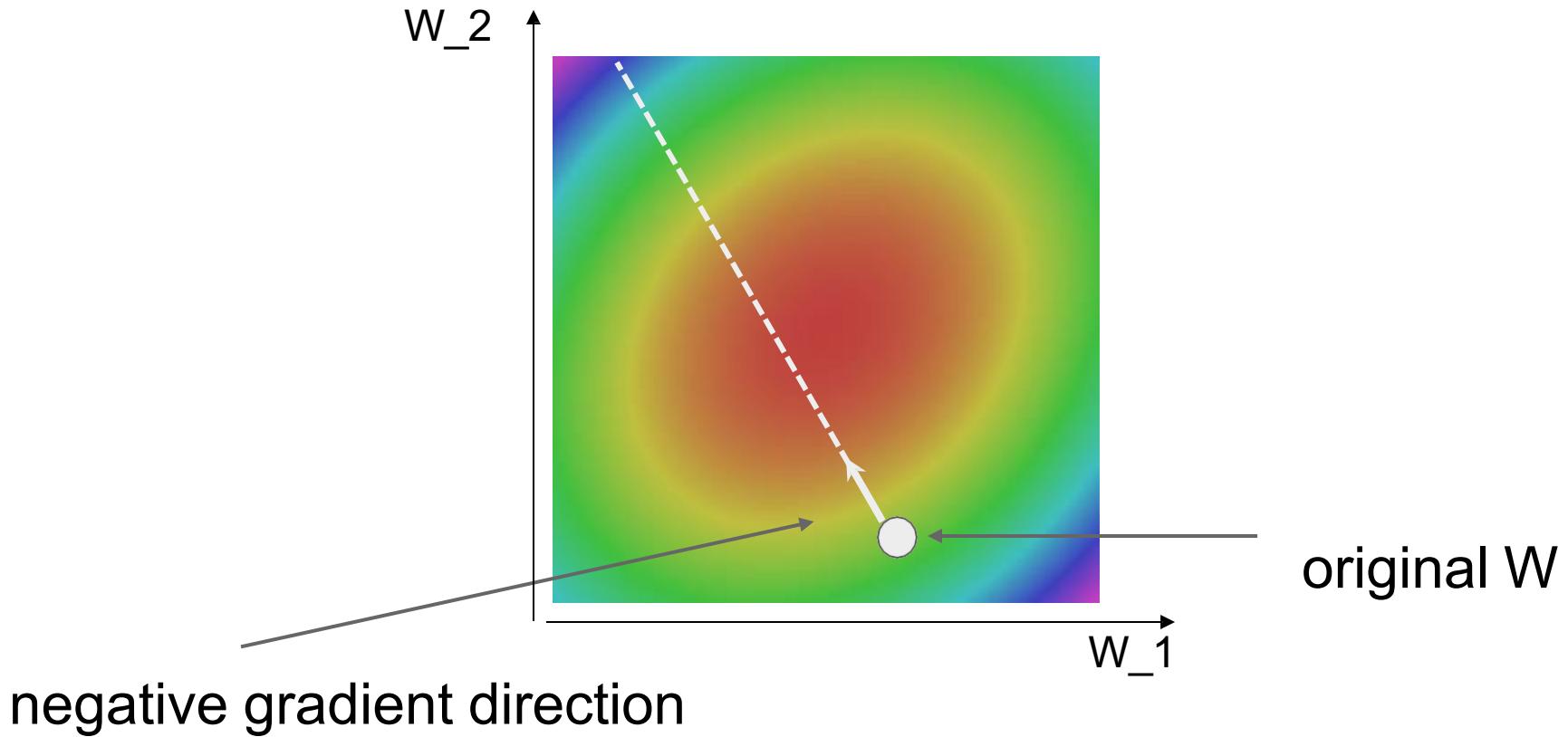
=>

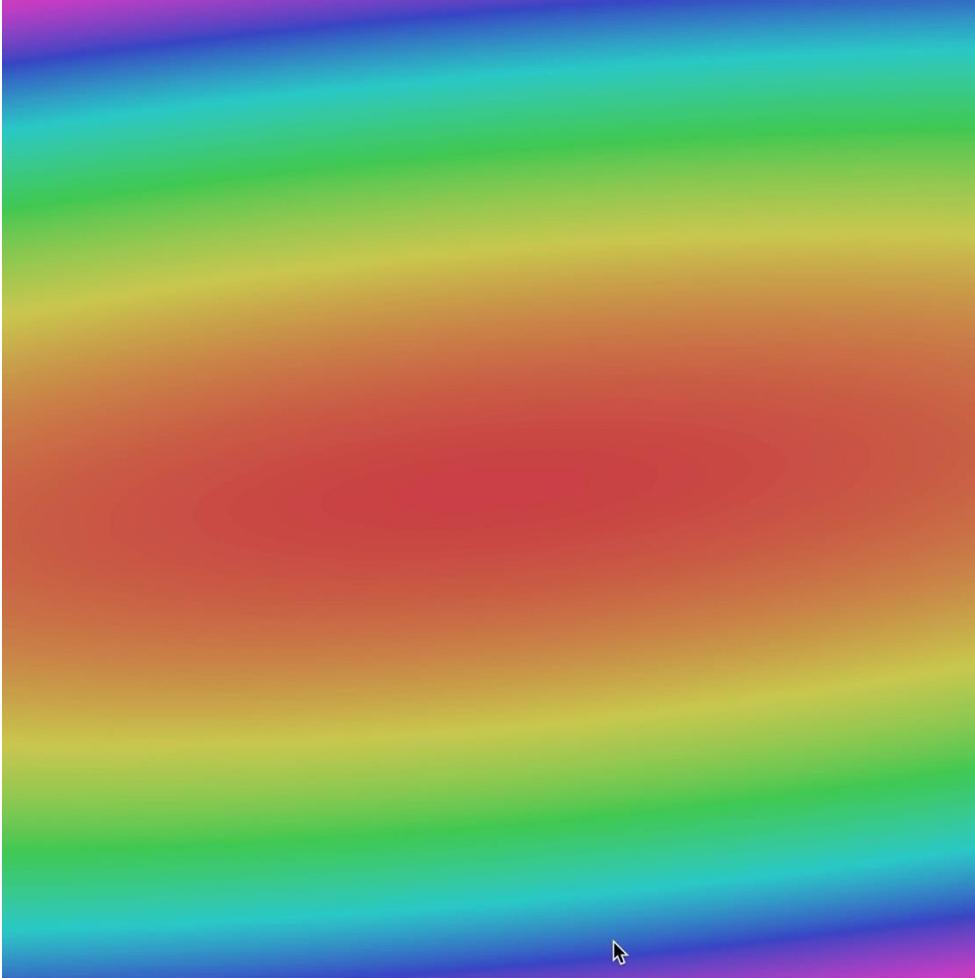
In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check**.

# Gradient Descent

```
# Vanilla Gradient Descent

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





# Stochastic Gradient Descent (SGD)

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

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

Full sum expensive  
when N is large!

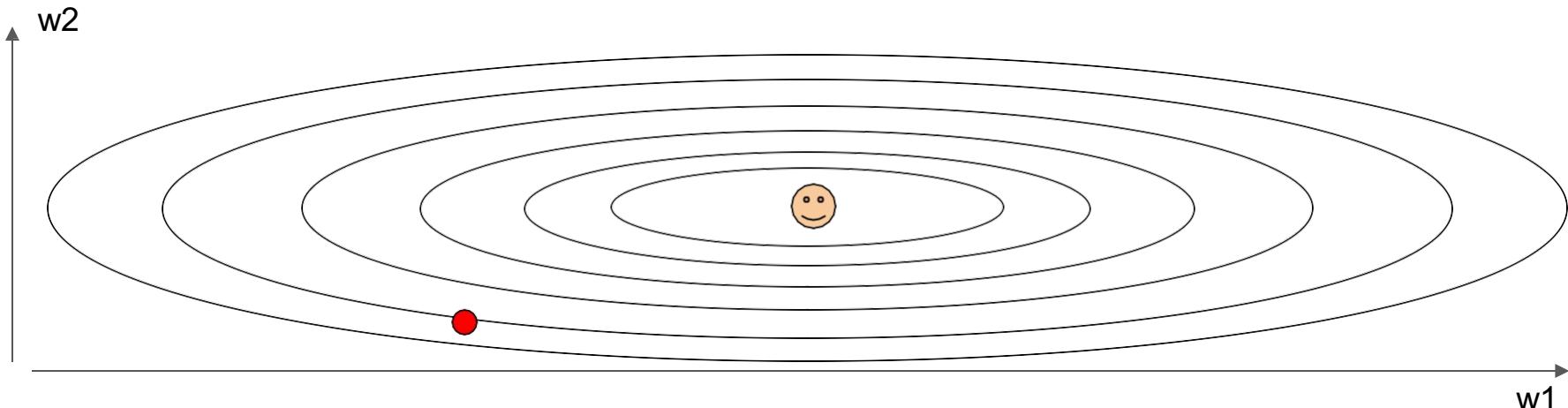
Approximate sum  
using a **minibatch** of  
examples  
32 / 64 / 128 common

```
# Vanilla Minibatch Gradient Descent
```

```
while True:  
    data_batch = sample_training_data(data, 256) # sample 256 examples  
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)  
    weights += - step_size * weights_grad # perform parameter update
```

# Optimization: Problem #1 with SGD

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

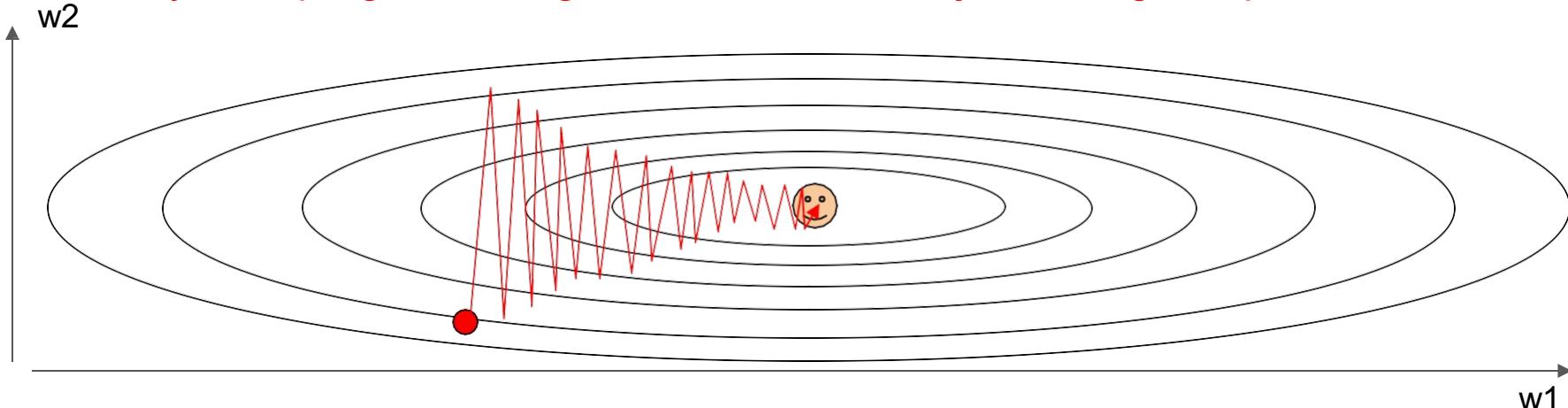


# Optimization: Problem #1 with SGD

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

What does gradient descent do?

Very slow progress along shallow dimension, jitter along steep direction

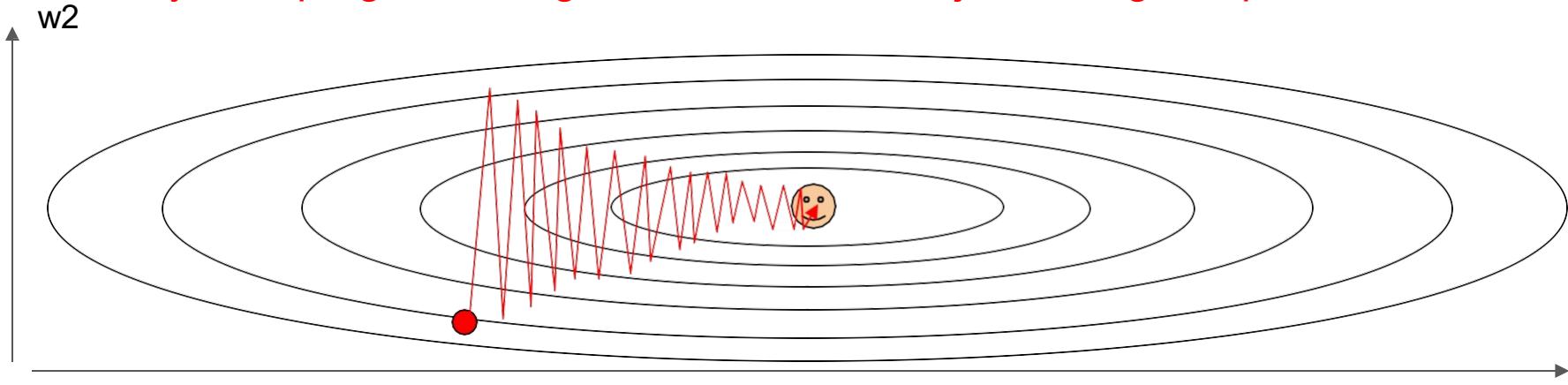


# Optimization: Problem #1 with SGD

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

What does gradient descent do?

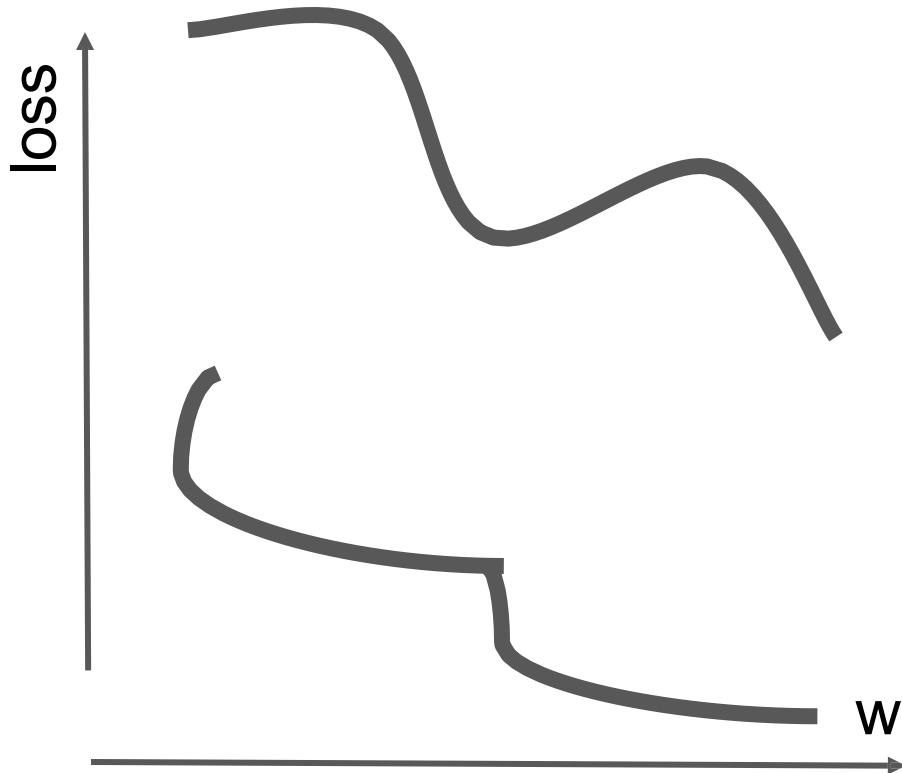
Very slow progress along shallow dimension, jitter along steep direction



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

# Optimization: Problem #2 with SGD

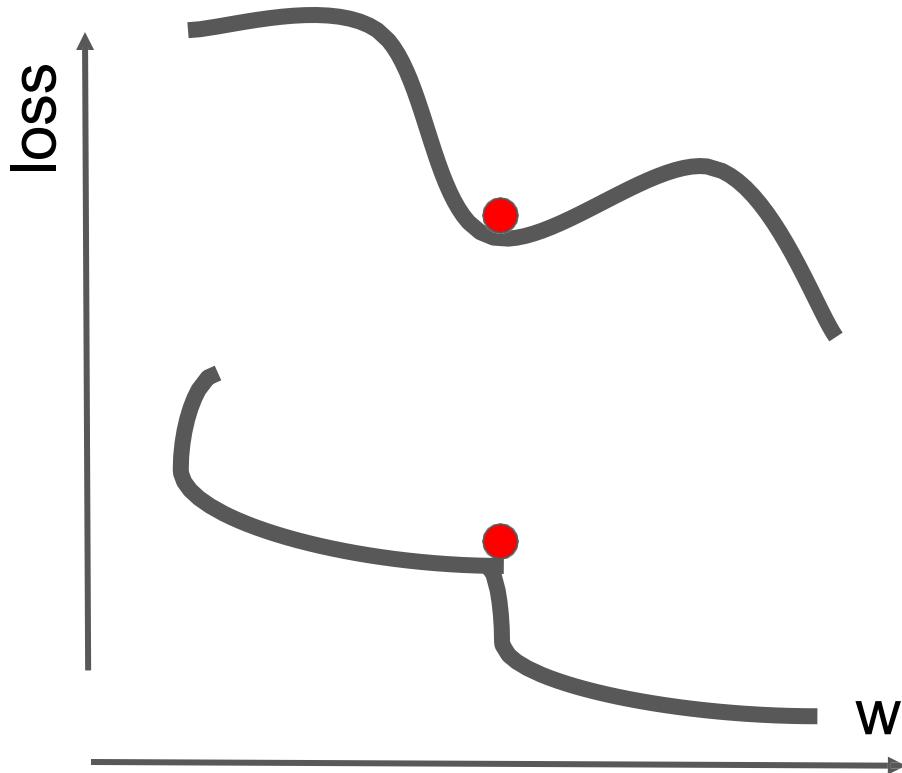
What if the loss  
function has a  
**local minima** or  
**saddle point**?



# Optimization: Problem #2 with SGD

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

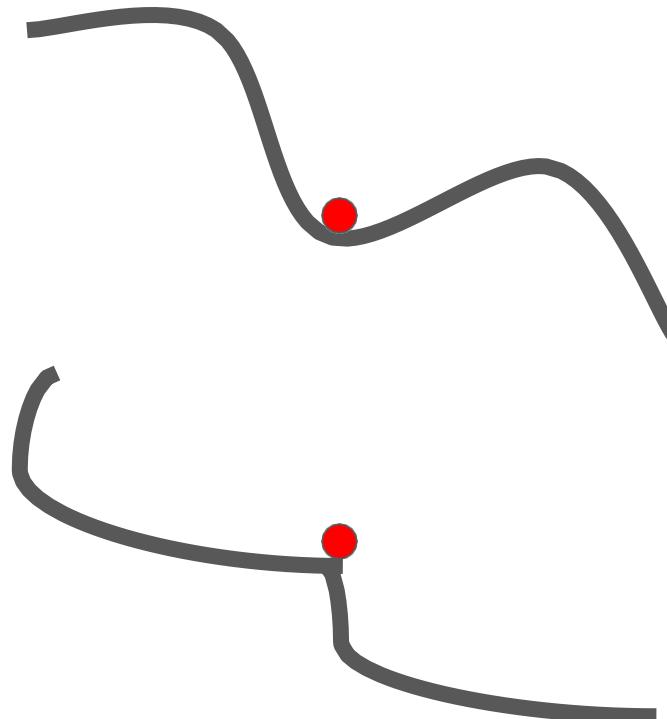
Zero gradient,  
gradient descent  
gets stuck



# Optimization: Problem #2 with SGD

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

Saddle points much  
more common in  
high dimension



# Optimization: Problem #2 with SGD

**saddle point** in two dimension

$$f(x, y) = x^2 - y^2$$

$$\frac{\partial}{\partial \textcolor{teal}{x}} (\textcolor{teal}{x}^2 - y^2) = 2x \rightarrow 2(\textcolor{teal}{0}) = 0$$

$$\frac{\partial}{\partial \textcolor{red}{y}} (x^2 - \textcolor{red}{y}^2) = -2y \rightarrow -2(\textcolor{red}{0}) = 0$$

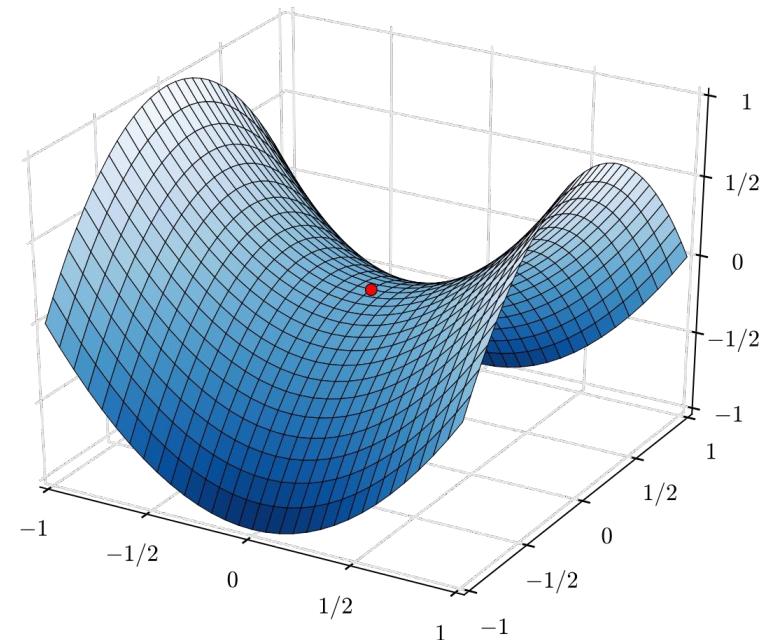


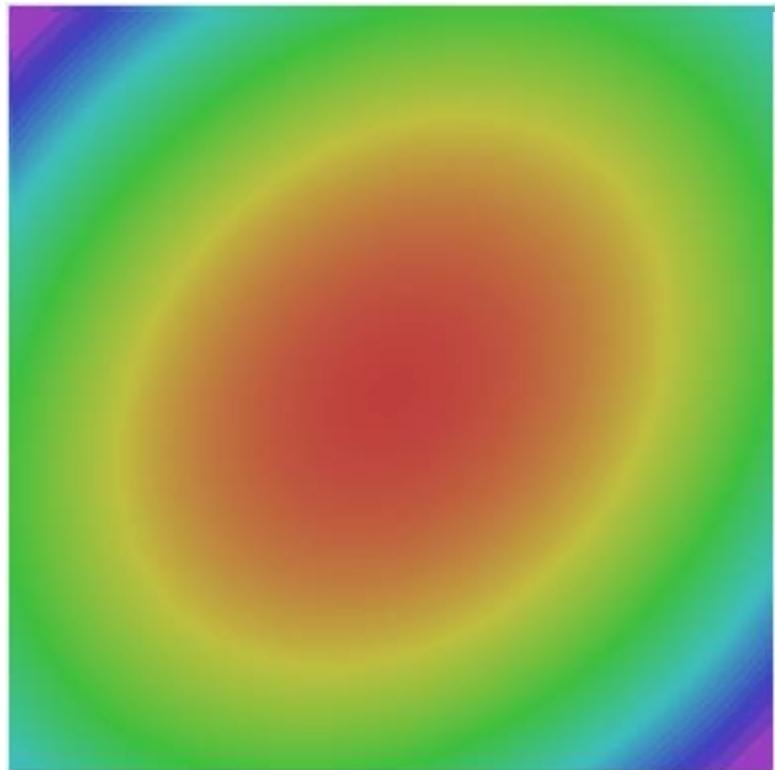
Image source: [https://en.wikipedia.org/wiki/Saddle\\_point](https://en.wikipedia.org/wiki/Saddle_point)

# Optimization: Problem #3 with SGD

Our gradients come from  
minibatches so they can be noisy!

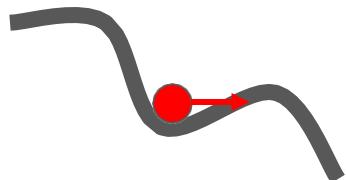
$$L(W) = \frac{1}{N} \sum_{i=1}^N L_i(x_i, y_i, W)$$

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

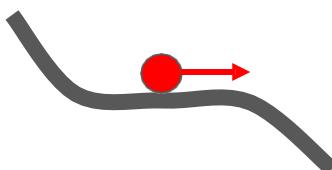


# SGD + Momentum

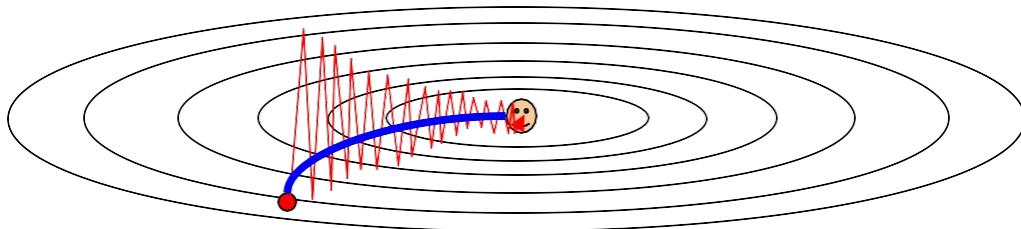
Local Minima



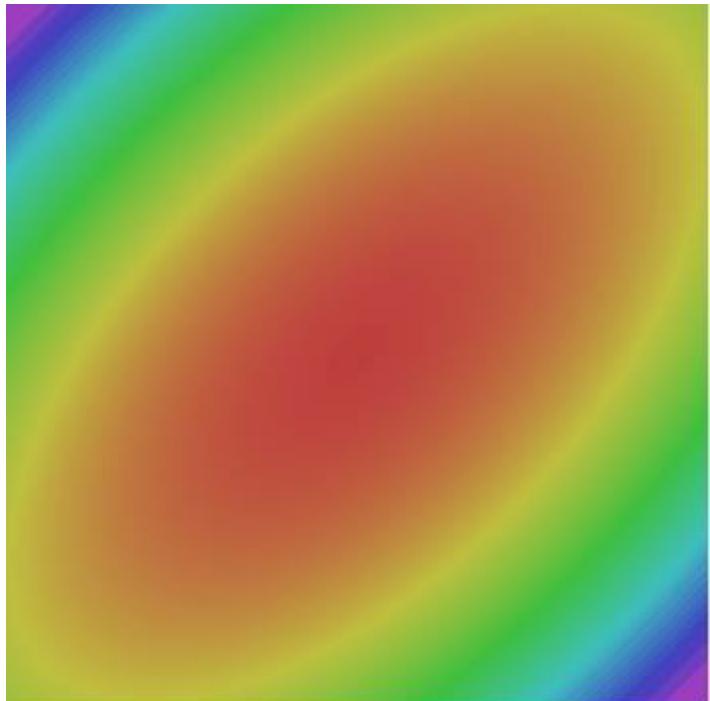
Saddle points



Poor Conditioning



# Gradient Noise



— SGD      — SGD+Momentum

# SGD: the simple two line update code

## SGD

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

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

# SGD + Momentum:

continue moving in the general direction as the previous iterations

SGD

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

SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

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

# SGD + Momentum:

continue moving in the general direction as the previous iterations

## SGD

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

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

## SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

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

# SGD + Momentum: alternative equivalent formulation

## SGD+Momentum

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

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

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

## SGD+Momentum

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

$$x_{t+1} = x_t - \alpha v_{t+1}$$

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

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

# More Complex Optimizers: RMSProp

SGD +  
Momentum

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

Adds element-wise scaling of the gradient based on the historical sum of squares in each dimension (with decay)



RMSProp

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

# More Complex Optimizers: RMSProp

SGD +  
Momentum

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

“Per-parameter learning rates”  
or “adaptive learning rates”



RMSProp

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

# RMSProp

## RMSProp

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

Q: What happens with RMSProp?

# RMSProp

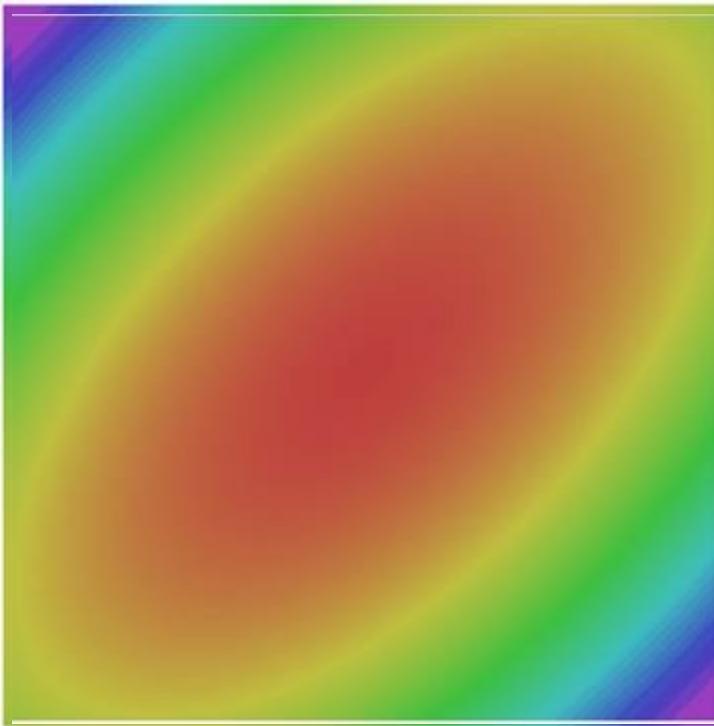
## RMSProp

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

Q: What happens with RMSProp?

Progress along “steep” directions is damped;  
progress along “flat” directions is accelerated

# RMSProp



- SGD
- SGD+Momentum
- RMSProp

# Optimizers: Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

# Adam (almost)

```
first_moment = 0
second_moment = 0
while True:
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Momentum

RMSProp

Sort of like RMSProp with momentum

Q: What happens at first timestep?

# Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

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

# Adam (full form)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

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

Adam with  $\text{beta1} = 0.9$ ,  
 $\text{beta2} = 0.999$ , and  $\text{learning\_rate} = 1\text{e-}3$  or  $5\text{e-}4$   
is a great starting point for many models!

# Adam



- SGD
- SGD+Momentum
- RMSProp
- Adam

# AdamW: Adam Variant with Weight Decay

Q: How does regularization interact  
with the optimizer? (e.g., L2)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

# AdamW: Adam Variant with Weight Decay

Q: How does regularization interact  
with the optimizer? (e.g., L2)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

A: It depends!

# AdamW: Adam Variant with Weight Decay

Q: How does regularization interact  
with the optimizer? (e.g., L2)

```
first_moment = 0      Standard Adam computes L2 here
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x) ←
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Used during moment  
calculations!

# AdamW: Adam Variant with Weight Decay

Q: How does regularization interact with the optimizer? (e.g., L2)

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x)
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdamW (Weight Decay) adds term here

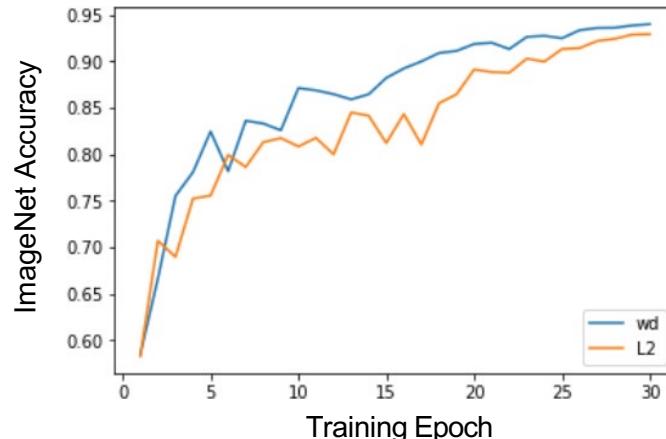
Computed after the moments!

# AdamW: Adam Variant with Weight Decay

Q: How does regularization interact with the optimizer? (e.g., L2)

```
first_moment = 0      Standard Adam computes L2 here
second_moment = 0
for t in range(1, num_iterations):
    dx = compute_gradient(x) ←
    first_moment = beta1 * first_moment + (1 - beta1) * dx
    second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
    first_unbias = first_moment / (1 - beta1 ** t)
    second_unbias = second_moment / (1 - beta2 ** t)
    x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

AdamW (Weight Decay) adds term here ↗



Source: <https://www.fast.ai/posts/2018-07-02-adam-weight-decay.html>

# Learning rate schedules

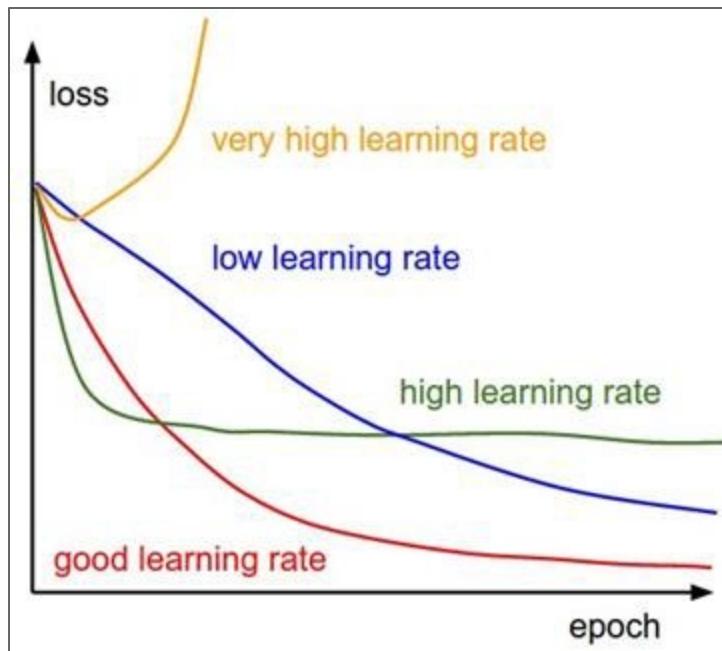
```
# Vanilla Gradient Descent

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



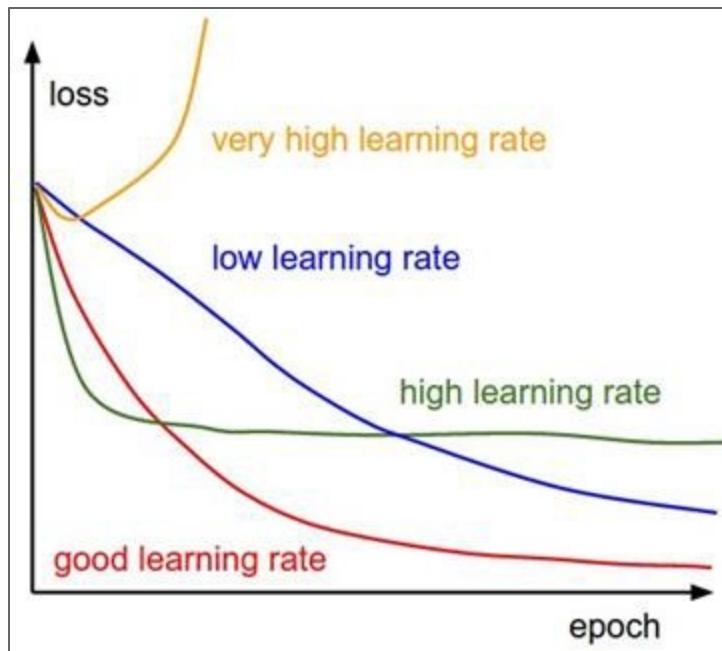
Learning rate

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



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

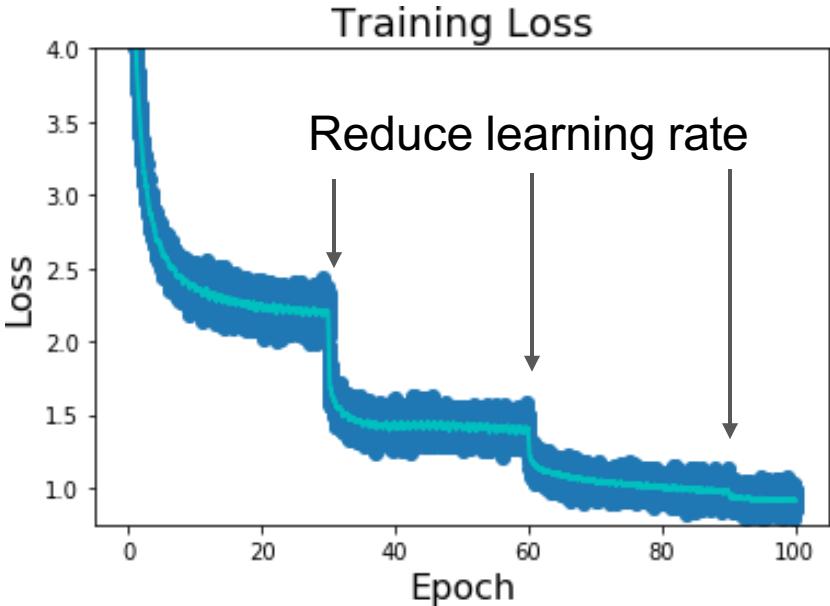
SGD, SGD+Momentum, RMSProp, Adam, AdamW all have **learning rate** as a hyperparameter.



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

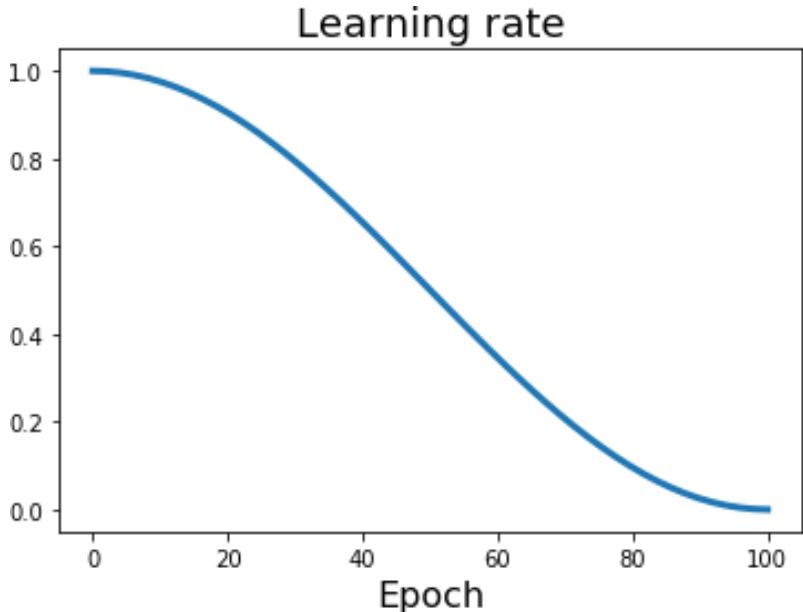
A: In reality, all of these could be good learning rates.

# Learning rate decays over time



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

# Learning Rate Decay



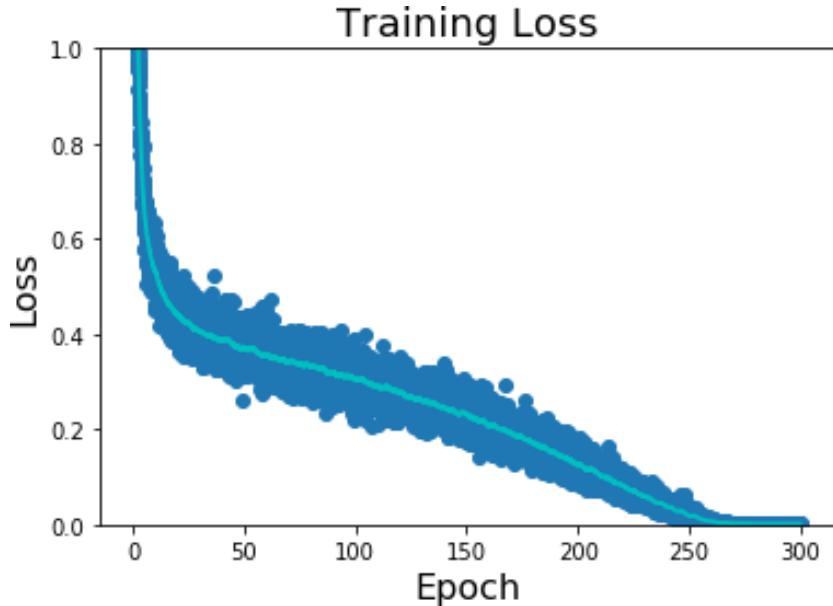
**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

- $\alpha_0$  : Initial learning rate
- $\alpha_t$  : Learning rate at epoch t
- $T$  : Total number of epochs

Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017  
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018  
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018  
Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

# Learning Rate Decay



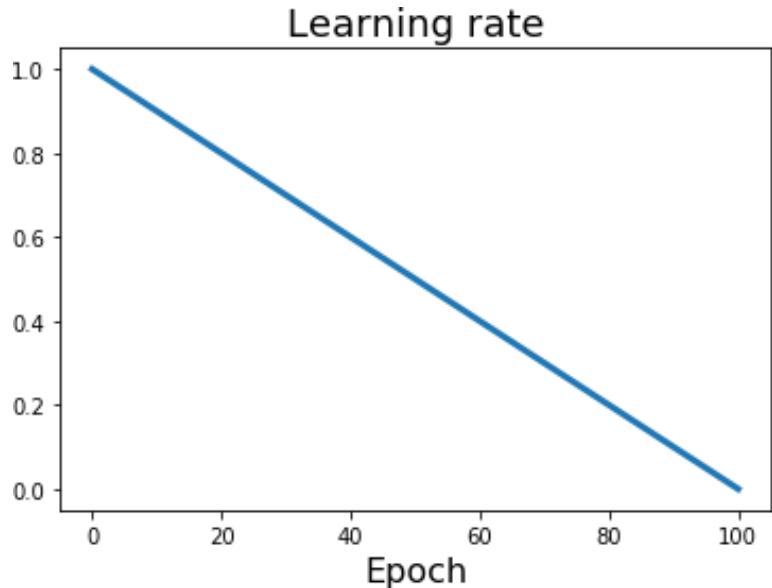
**Step:** Reduce learning rate at a few fixed points. E.g. for ResNets, multiply LR by 0.1 after epochs 30, 60, and 90.

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

- Loshchilov and Hutter, "SGDR: Stochastic Gradient Descent with Warm Restarts", ICLR 2017  
Radford et al, "Improving Language Understanding by Generative Pre-Training", 2018  
Feichtenhofer et al, "SlowFast Networks for Video Recognition", arXiv 2018  
Child et al, "Generating Long Sequences with Sparse Transformers", arXiv 2019

- $\alpha_0$  : Initial learning rate  
 $\alpha_t$  : Learning rate at epoch t  
 $T$  : Total number of epochs

# Learning Rate Decay



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

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

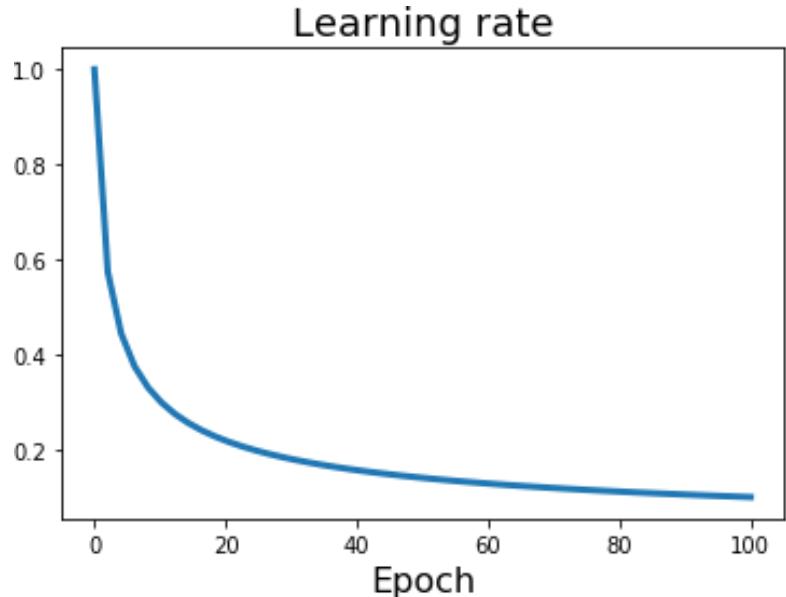
**Linear:**  $\alpha_t = \alpha_0(1 - t/T)$

$\alpha_0$  : Initial learning rate

$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

# Learning Rate Decay



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

**Cosine:**  $\alpha_t = \frac{1}{2}\alpha_0(1 + \cos(t\pi/T))$

**Linear:**  $\alpha_t = \alpha_0(1 - t/T)$

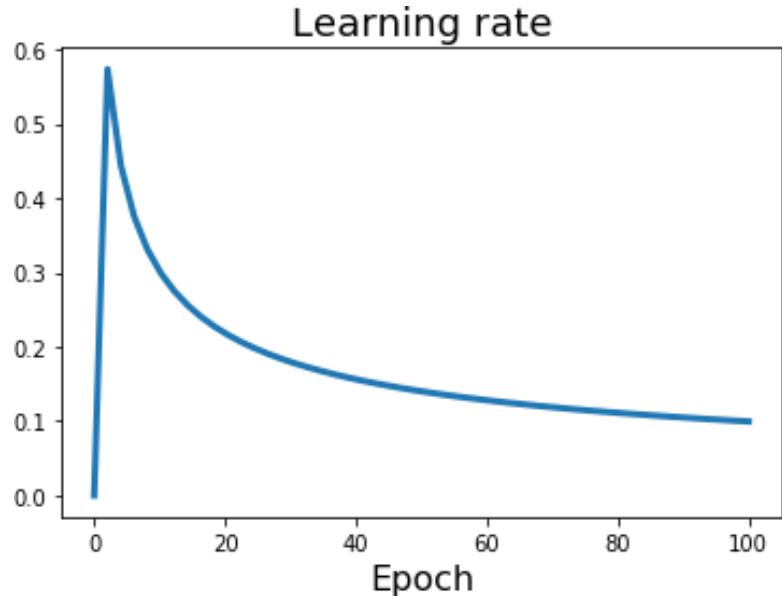
**Inverse sqrt:**  $\alpha_t = \alpha_0/\sqrt{t}$

$\alpha_0$  : Initial learning rate

$\alpha_t$  : Learning rate at epoch  $t$

$T$  : Total number of epochs

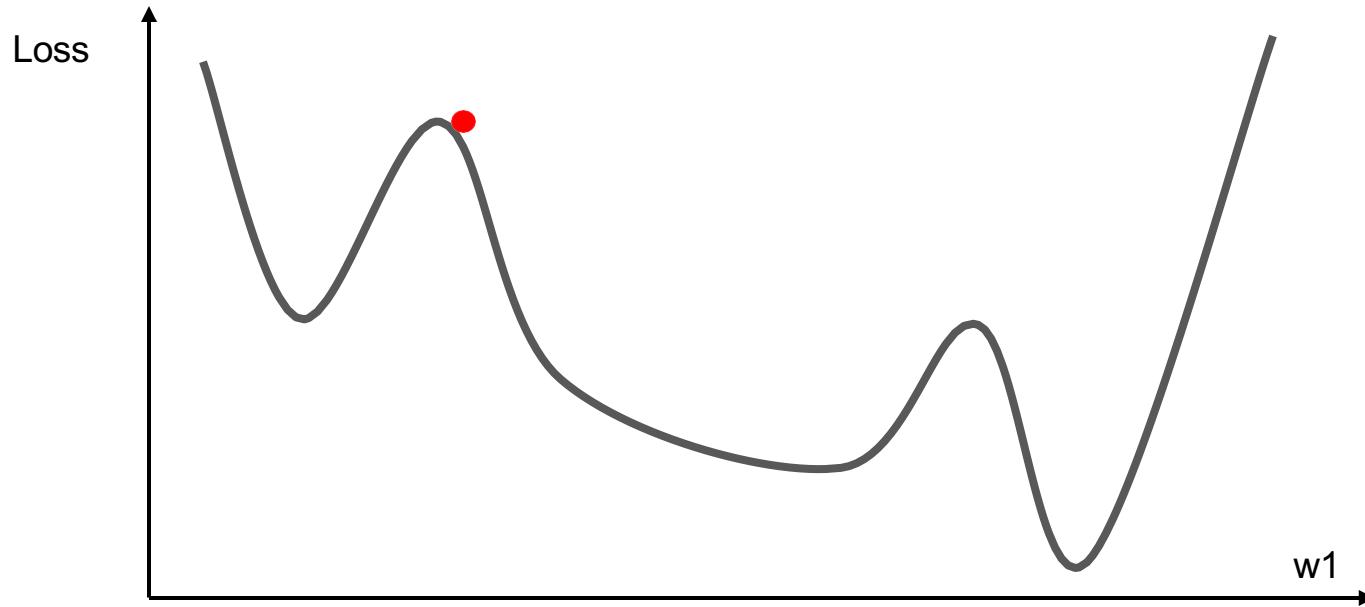
# Learning Rate Decay: Linear Warmup



High initial learning rates can make loss explode; linearly increasing learning rate from 0 over the first ~5,000 iterations can prevent this.

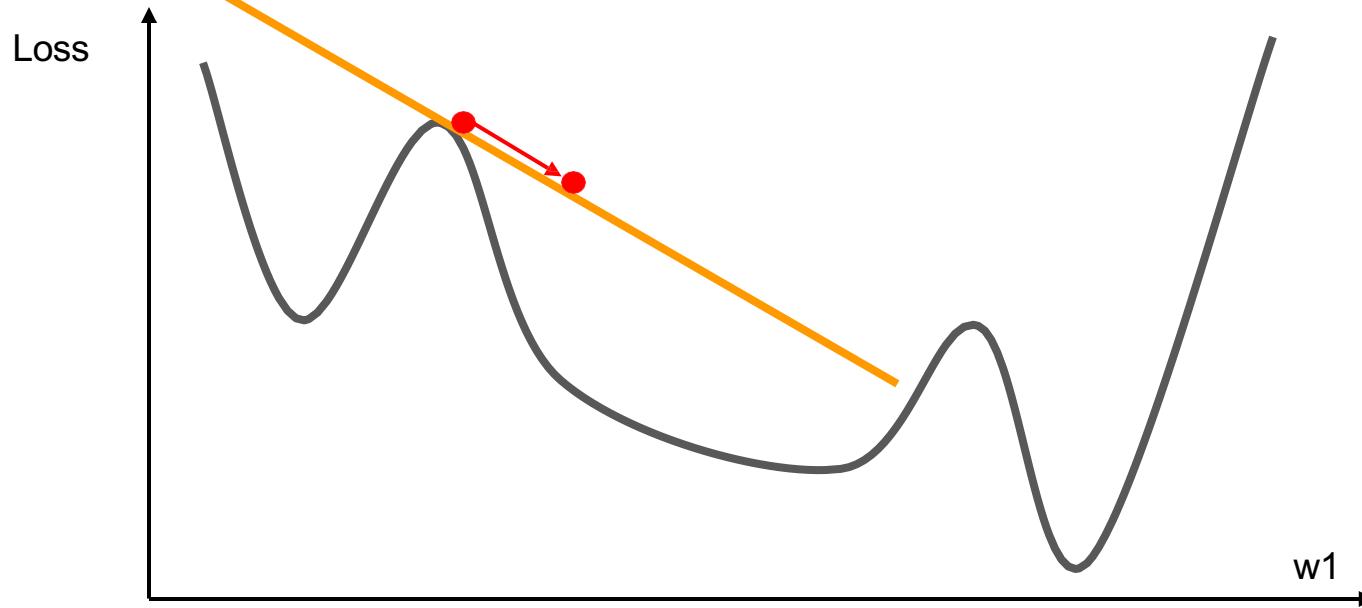
Empirical rule of thumb: If you increase the batch size by  $N$ , also scale the initial learning rate by  $N$

# First-Order Optimization



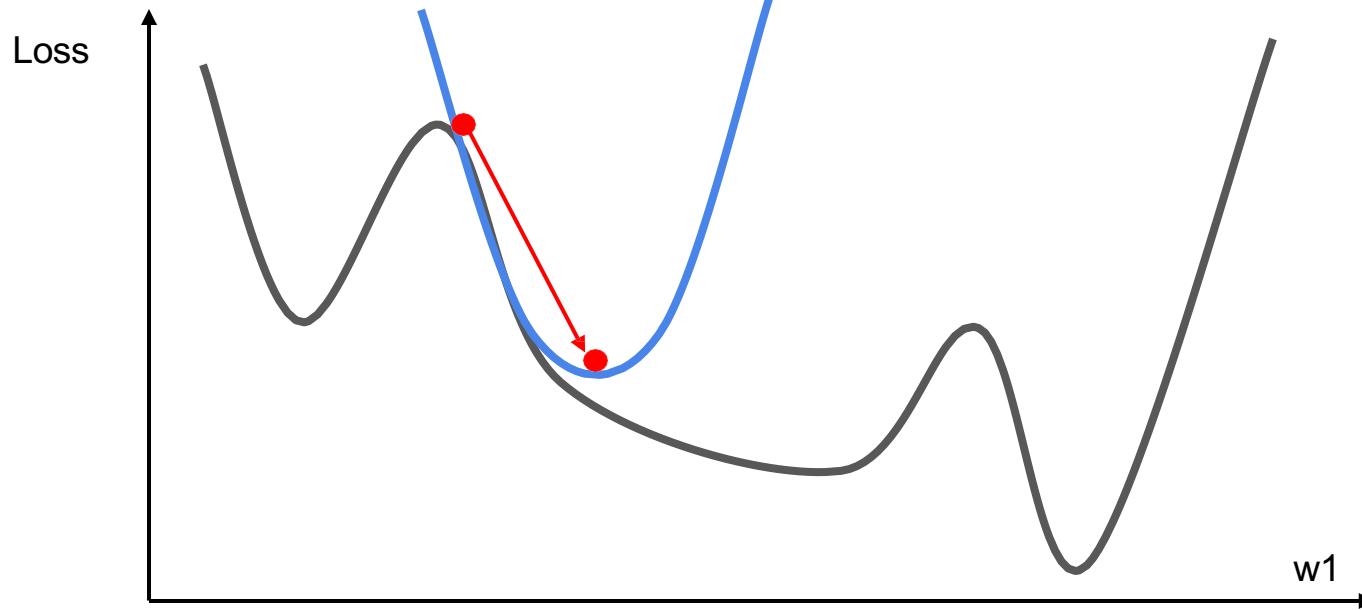
# First-Order Optimization

- (1) Use gradient form linear approximation
- (2) Step to minimize the approximation



# Second-Order Optimization

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



# Second-Order Optimization

second-order Taylor expansion:

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

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

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

Q: Why is this bad for deep learning?

# Second-Order Optimization

second-order Taylor expansion:

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

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

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

Hessian has  $O(N^2)$  elements

Inverting takes  $O(N^3)$

$N = (\text{Tens or Hundreds of}) \text{ Millions}$

Q: Why is this bad for deep learning?

# In practice:

- **Adam(W)** is a good default choice in many cases; it often works ok even with constant learning rate
- **SGD+Momentum** can outperform Adam but may require more tuning of LR and schedule
- If you can afford to do full batch updates then look beyond 1<sup>st</sup> order optimization (**2<sup>nd</sup> order and beyond**)

# Looking Ahead: How to optimize more complex functions?

(Currently) Linear score function:  $f = Wx$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

# Neural networks: 2 layers

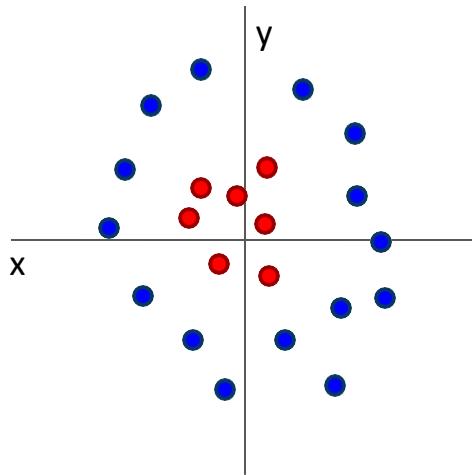
(Currently) Linear score function:  $f = Wx$

(Next Class) 2-layer Neural Network  $f = W_2 \max(0, W_1 x)$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

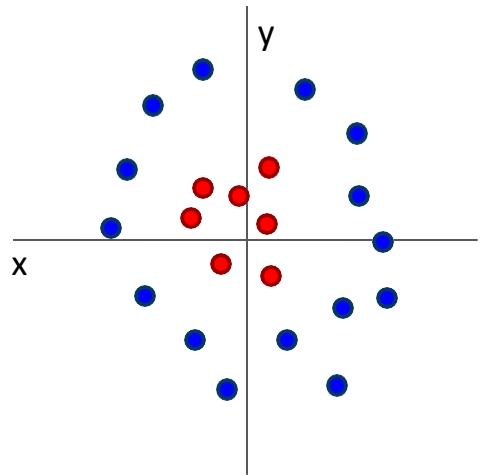
(In practice we will usually add a learnable bias at each layer as well)

# Why do we want non-linearity?

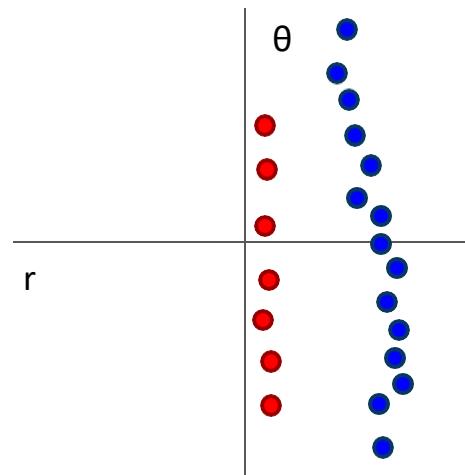


Cannot separate red and  
blue points with linear  
classifier

# Why do we want non-linearity?



$$f(x, y) = (r(x, y), \theta(x, y))$$



Cannot separate red and blue points with linear classifier

After applying feature transform, points can be separated by linear classifier

# Neural Networks and Backpropagation

# Recap

- We have some dataset of  $(x, y)$
- We have a score function:
- We have a loss function:

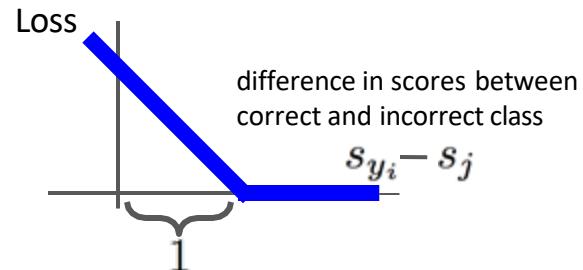
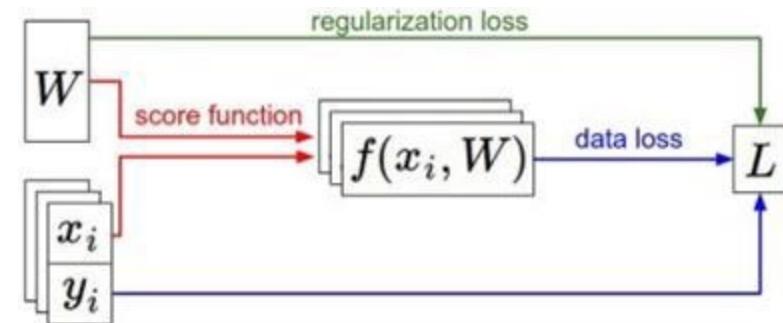
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right) \text{ Softmax}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + R(W) \text{ Full loss}$$

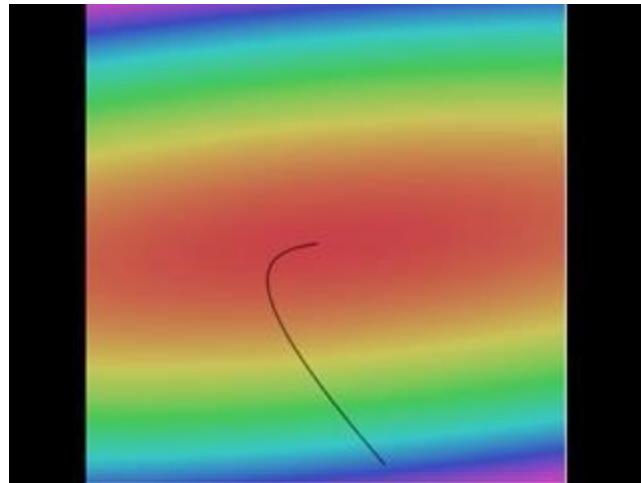
$$\begin{aligned} L_i &= \sum_{j \neq y_i} \begin{cases} 0 & \text{if } s_{y_i} \geq s_j + 1 \\ s_j - s_{y_i} + 1 & \text{otherwise} \end{cases} \\ &= \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \end{aligned}$$

SVM/hinge Loss (refer to  
Lecture 2 reading assignment )

e.g.  
 $s = f(x; W) = Wx$



# Finding the best W: Optimize with Gradient Descent



```
# Vanilla Gradient Descent

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

[Landscape image](#) is CC0 1.0 public domain

[Walking man image](#) is CC0 1.0 public domain

# Gradient descent

$$\frac{df(x)}{dx} = \lim_{h \rightarrow 0} \frac{f(x + h) - f(x)}{h}$$

Numerical gradient: slow ☹, approximate ☹, easy to write ☺  
Analytic gradient: fast ☺, exact ☺, error-prone ☹

In practice: Derive analytic gradient, check your implementation with numerical gradient

# Stochastic Gradient Descent (SGD)

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

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

Full sum is expensive  
when N is large!

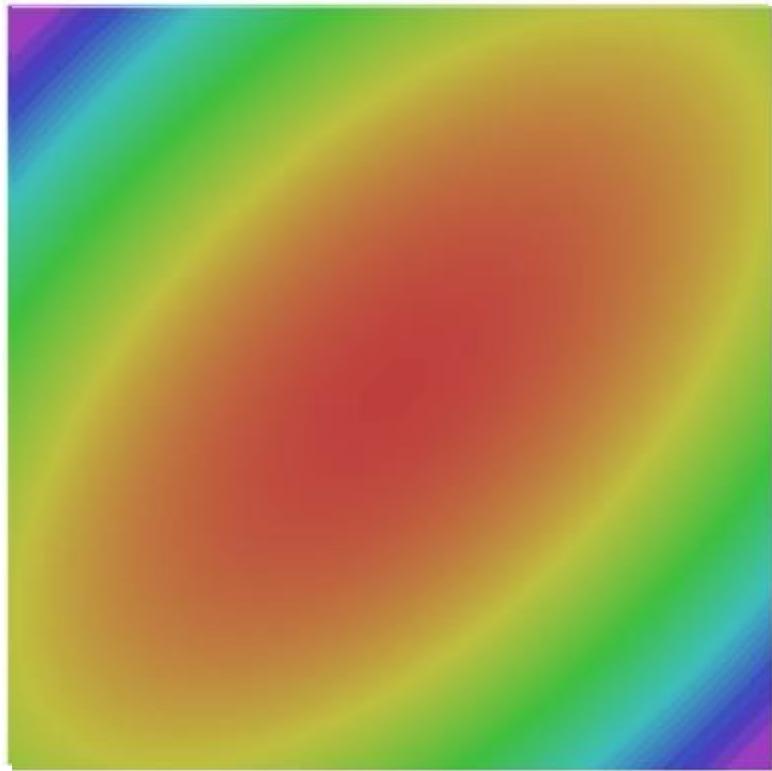
Approximate sum using a  
minibatch of examples  
32 / 64 / 128 / 256

```
# Vanilla Minibatch Gradient Descent
```

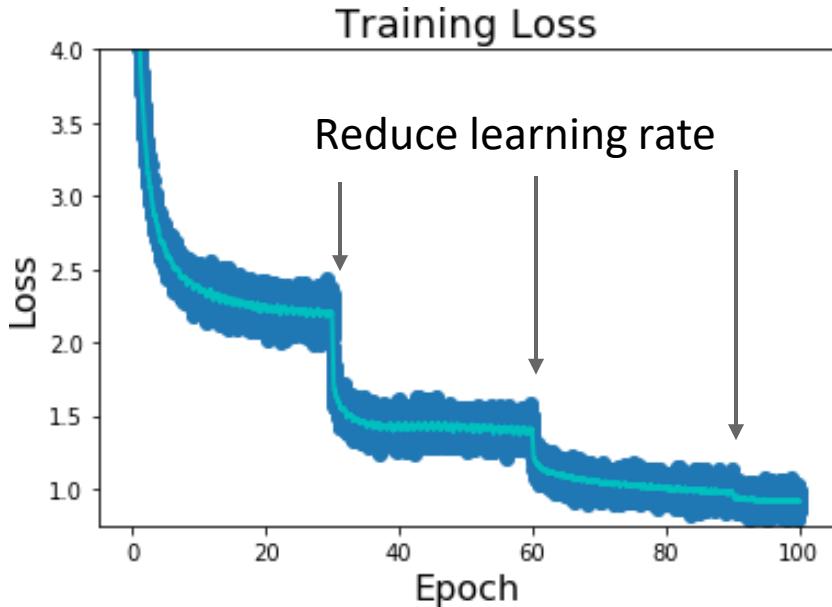
```
while True:
    data_batch = sample_training_data(data, 256) # sample 256 examples
    weights_grad = evaluate_gradient(loss_fun, data_batch, weights)
    weights += - step_size * weights_grad # perform parameter update
```

# Last time: fancy optimizers



- SGD
- SGD+Momentum
- RMSProp
- Adam

# Last time: learning rate scheduling



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

Cosine:  $\alpha_t = \frac{1}{2}\alpha_0 (1 + \cos(t\pi/T))$

Linear:  $\alpha_t = \alpha_0(1 - t/T)$

Inverse sqrt:  $\alpha_t = \alpha_0/\sqrt{t}$

$\alpha_0$  : Initial learning rate

$\alpha_t$  : Learning rate at epoch t

$T$  : Total number of epochs

# Neural Networks

# Neural networks: the original linear classifier

(Before) Linear score function:

$$f = Wx$$

$$x \in \mathbb{R}^D, W \in \mathbb{R}^{C \times D}$$

# Neural networks: 2 layers

(Before) Linear score function:

$$f = Wx$$

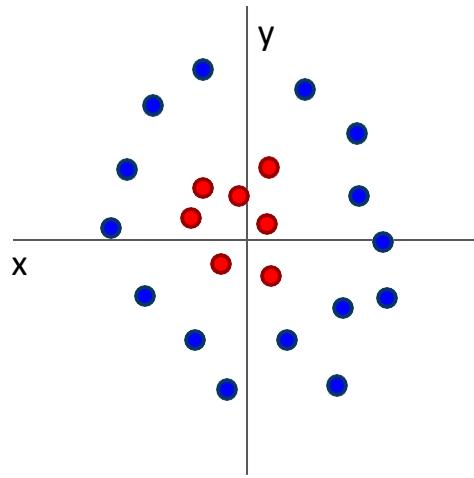
(Now) 2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

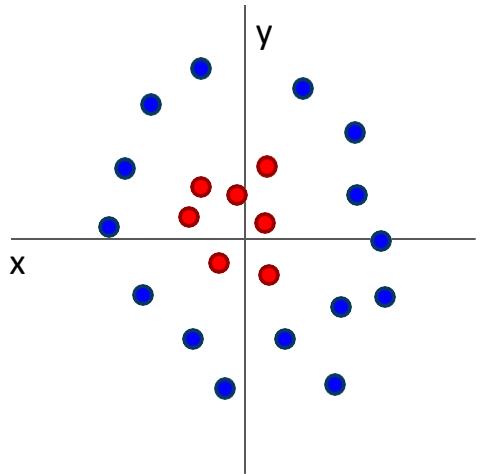
(In practice we will usually add a learnable bias at each layer as well)

# Why do we want non-linearity?

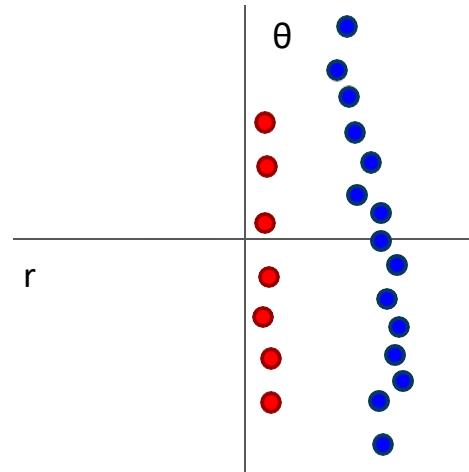


Cannot separate red and  
blue points with linear  
classifier

# Why do we want non-linearity?



$$f(x, y) = (r(x, y), \theta(x, y))$$



Cannot separate red and blue points with linear classifier

After applying feature transform, points can be separated by linear classifier

# Neural networks: also called fully connected network

(Before) Linear score function:

$$f = Wx$$

(Now) 2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

“Neural Network” is a very broad term; these are more accurately called “fully-connected networks” or sometimes “multi-layer perceptrons” (MLP)

(In practice we will usually add a learnable bias at each layer as well)

# Neural networks: 3 layers

(Before) Linear score function:

$$f = Wx$$

(Now) 2-layer Neural Network  
or 3-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

$$f = W_3 \max(0, W_2 \max(0, W_1 x))$$

$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H_1 \times D}, W_2 \in \mathbb{R}^{H_2 \times H_1}, W_3 \in \mathbb{R}^{C \times H_2}$$

(In practice we will usually add a learnable bias at each layer as well)

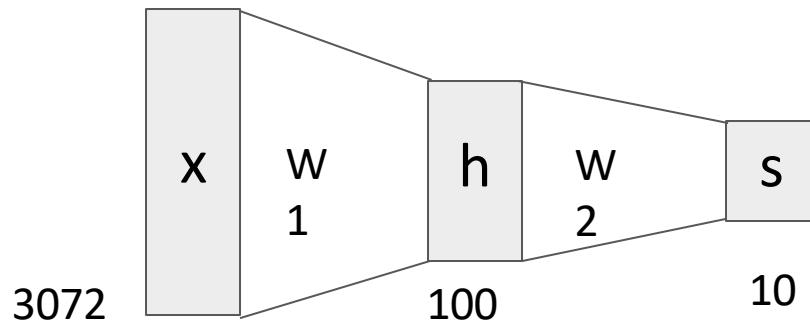
# Neural networks: hierarchical computation

(Before) Linear score function:

$$f = Wx$$

(Now) 2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$



$$x \in \mathbb{R}^D, W_1 \in \mathbb{R}^{H \times D}, W_2 \in \mathbb{R}^{C \times H}$$

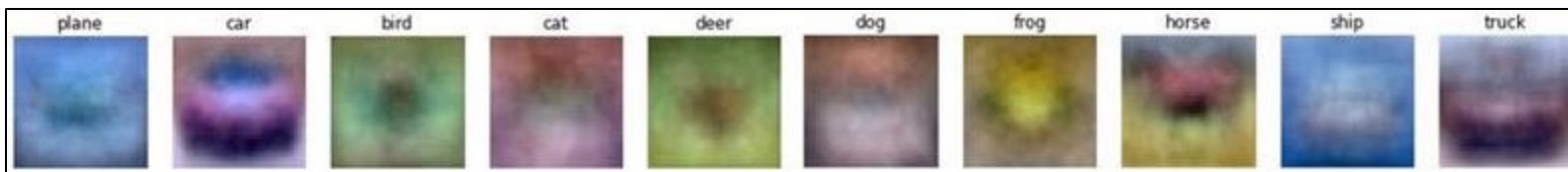
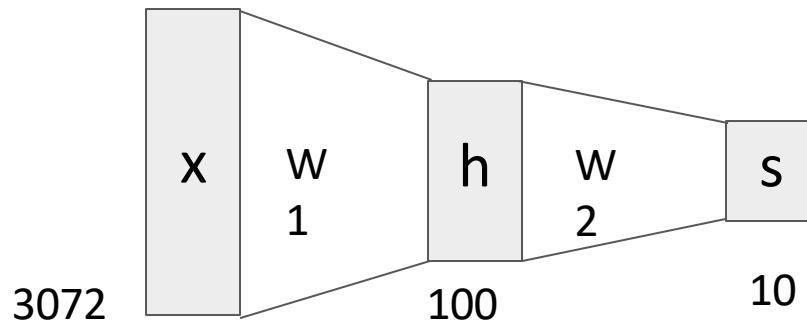
# Neural networks: learning 100s of templates

(Before) Linear score function:

$$f = Wx$$

(Now) 2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$



Learn 100 templates instead of 10.

Share templates between classes

# Neural networks: why is max operator important?

(Before) Linear score function:

$$f = Wx$$

(Now) 2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

The function  $\max(0, z)$  is called the activation function.

Q: What if we try to build a neural network without one?

$$f = W_2 W_1 x$$

# Neural networks: why is max operator important?

(Before) Linear score function:

$$f = Wx$$

(Now) 2-layer Neural Network

$$f = W_2 \max(0, W_1 x)$$

The function  $\max(0, z)$  is called the activation function.

Q: What if we try to build a neural network without one?

$$f = W_2 W_1 x \quad W_3 = W_2 W_1 \in \mathbb{R}^{C \times H}, f = W_3 x$$

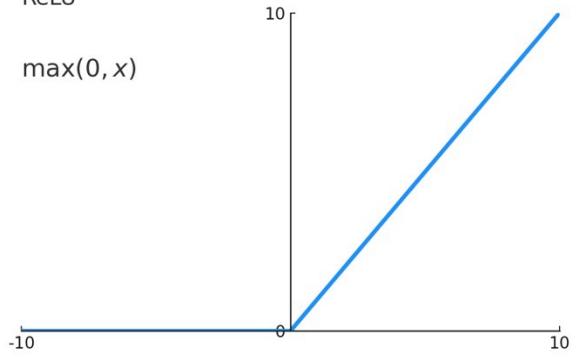
A: We end up with a linear classifier again!

# Activation functions

ReLU is a good default choice for most problems

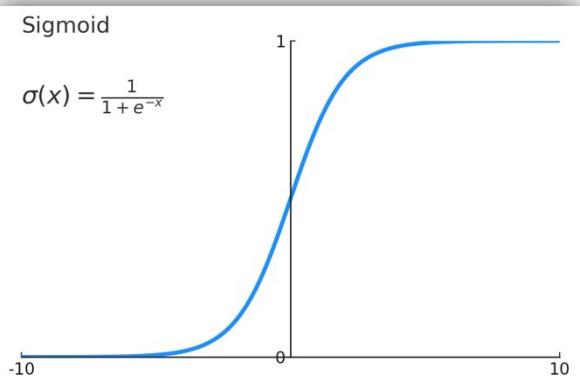
ReLU

$$\max(0, x)$$



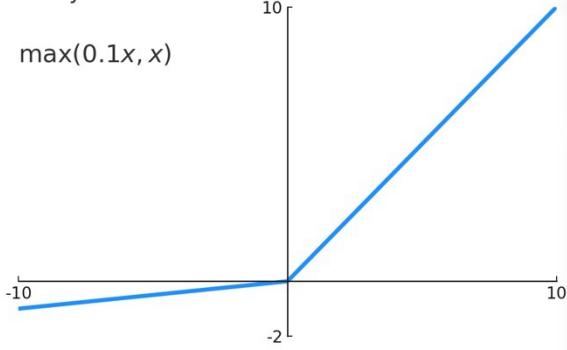
Sigmoid

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



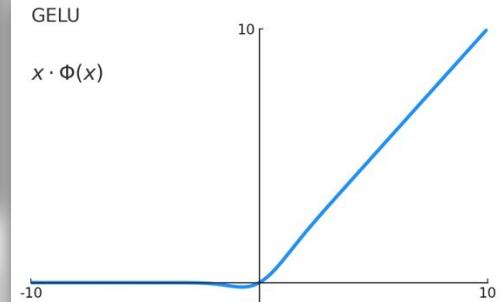
Leaky ReLU

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



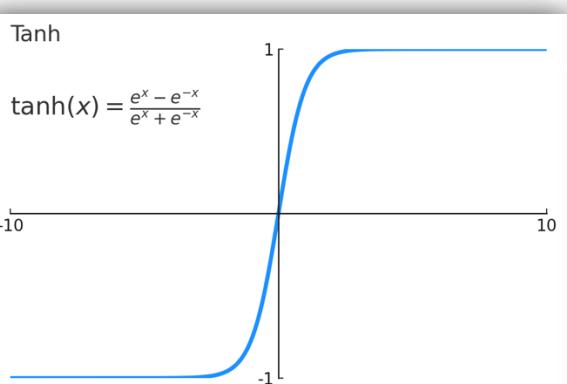
GELU

$$x \cdot \Phi(x)$$



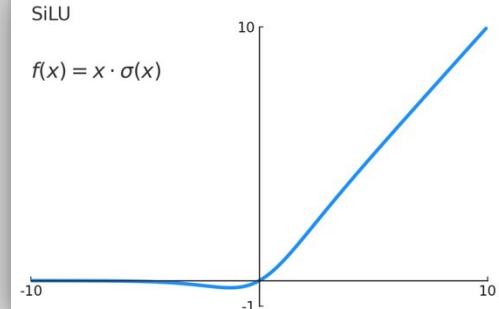
Tanh

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



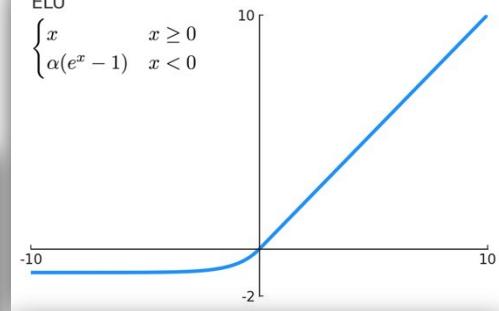
SiLU

$$f(x) = x \cdot \sigma(x)$$

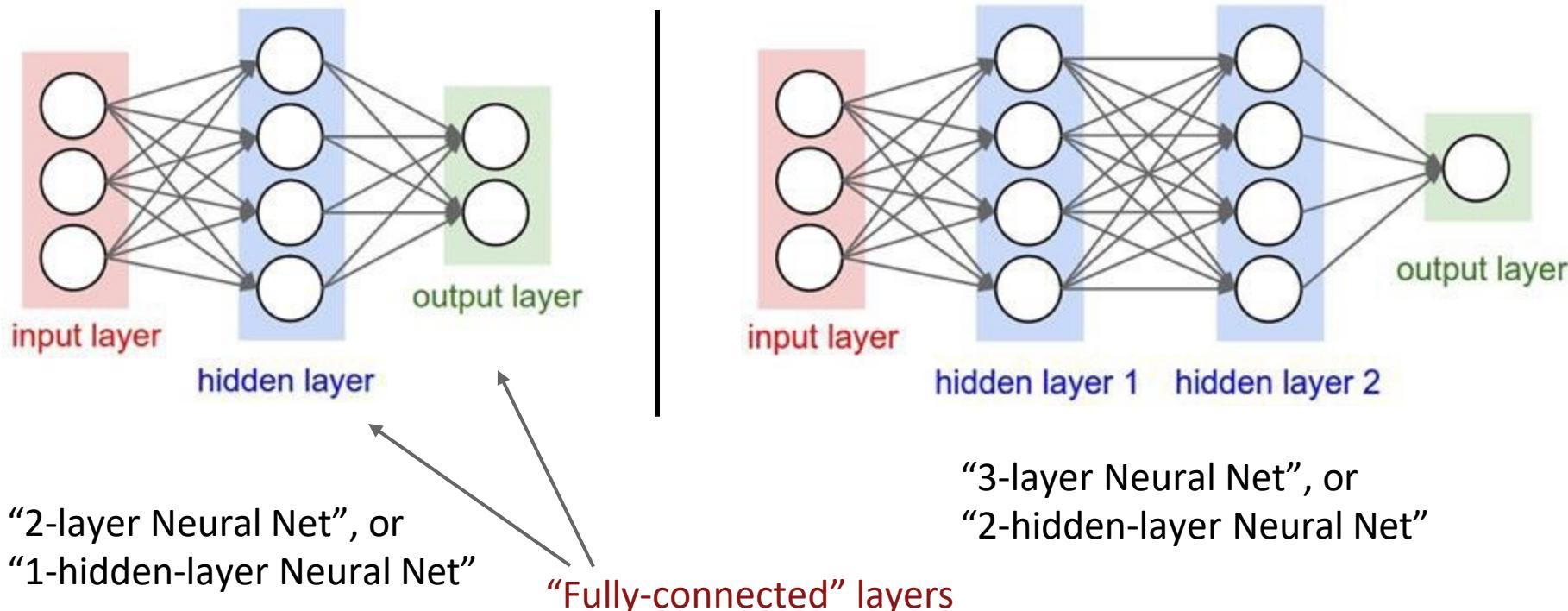


ELU

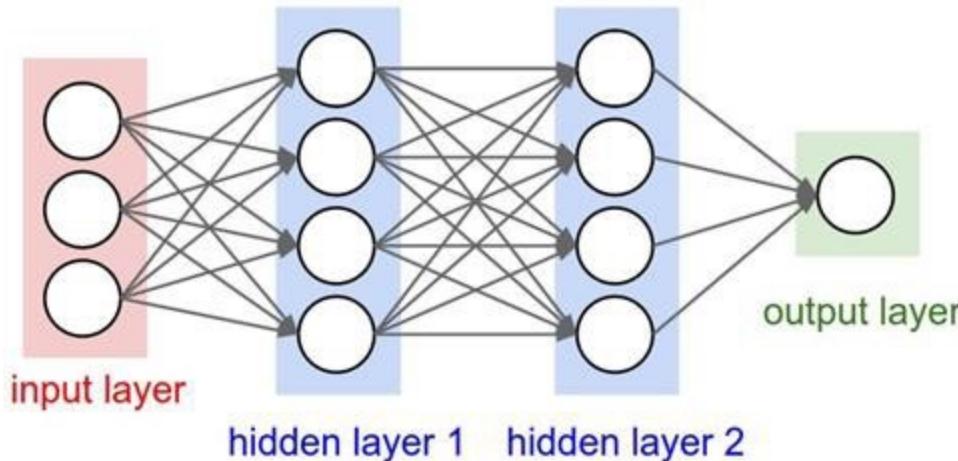
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



# Neural networks: Architectures



# Example feed-forward computation of a neural network



```
# forward-pass of a 3-layer neural network:  
f = lambda x: 1.0/(1.0 + np.exp(-x)) # activation function (use sigmoid)  
x = np.random.randn(3, 1) # random input vector of three numbers (3x1)  
h1 = f(np.dot(W1, x) + b1) # calculate first hidden layer activations (4x1)  
h2 = f(np.dot(W2, h1) + b2) # calculate second hidden layer activations (4x1)  
out = np.dot(W3, h2) + b3 # output neuron (1x1)
```

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14 grad_y_pred = 2.0 * (y_pred - y)
15 grad_w2 = h.T.dot(grad_y_pred)
16 grad_h = grad_y_pred.dot(w2.T)
17 grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19 w1 -= 1e-4 * grad_w1
20 w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

Define the network

Forward pass

Calculate the analytical gradients

# Full implementation of training a 2-layer Neural Network needs ~20 lines:

```
1 import numpy as np
2 from numpy.random import randn
3
4 N, D_in, H, D_out = 64, 1000, 100, 10
5 x, y = randn(N, D_in), randn(N, D_out)
6 w1, w2 = randn(D_in, H), randn(H, D_out)
7
8 for t in range(2000):
9     h = 1 / (1 + np.exp(-x.dot(w1)))
10    y_pred = h.dot(w2)
11    loss = np.square(y_pred - y).sum()
12    print(t, loss)
13
14    grad_y_pred = 2.0 * (y_pred - y)
15    grad_w2 = h.T.dot(grad_y_pred)
16    grad_h = grad_y_pred.dot(w2.T)
17    grad_w1 = x.T.dot(grad_h * h * (1 - h))
18
19    w1 -= 1e-4 * grad_w1
20    w2 -= 1e-4 * grad_w2
```

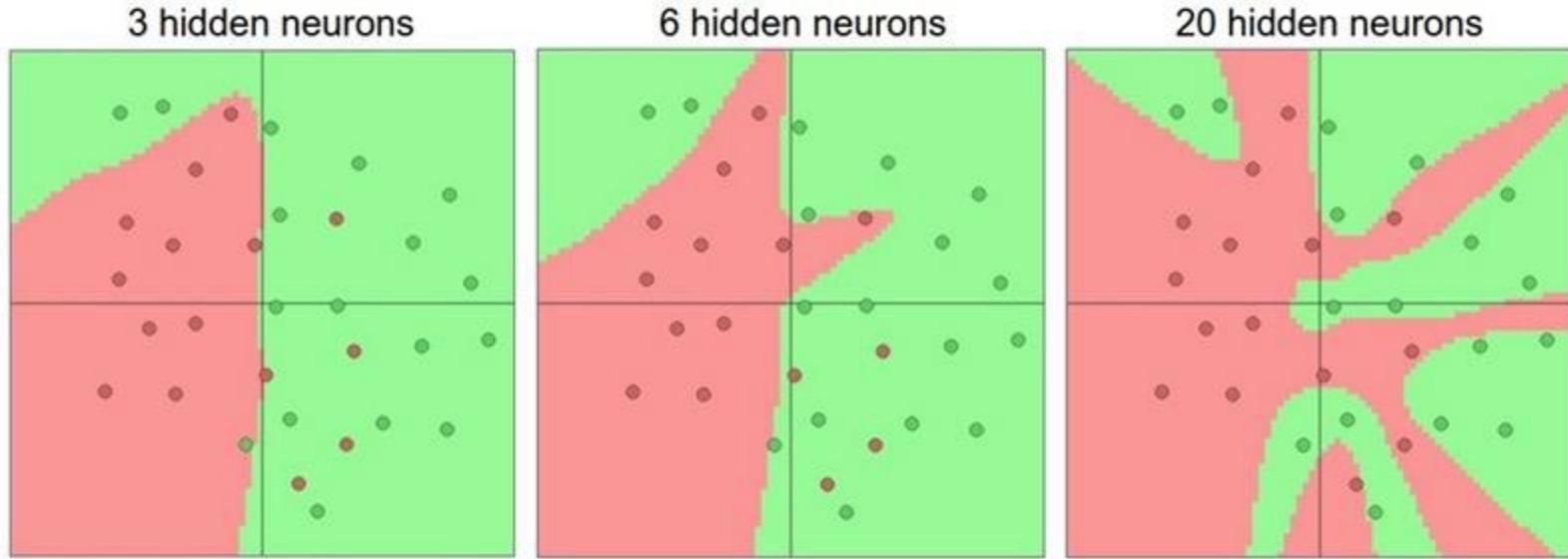
Define the network

Forward pass

Calculate the analytical gradients

Gradient descent

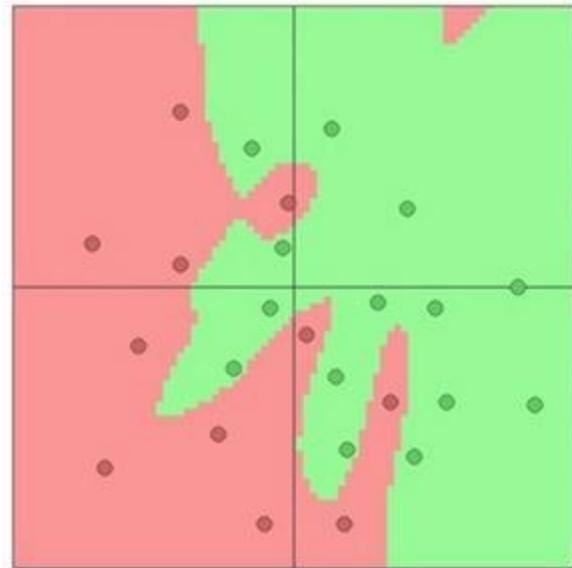
# Setting the number of layers and their sizes



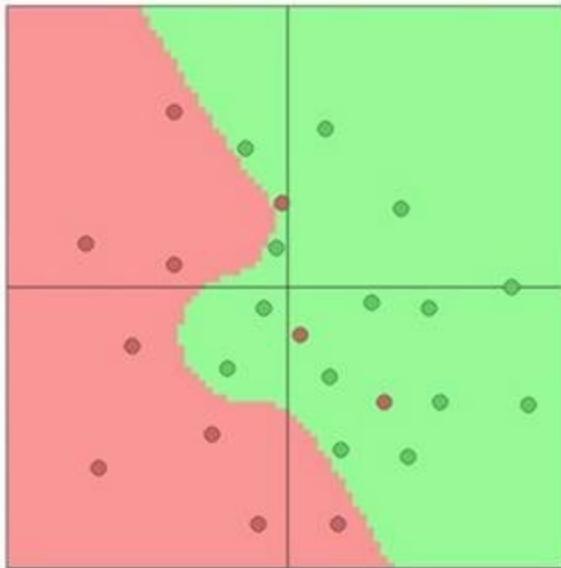
more neurons = more capacity

Do not use size of neural network as a regularizer. Use stronger regularization instead:

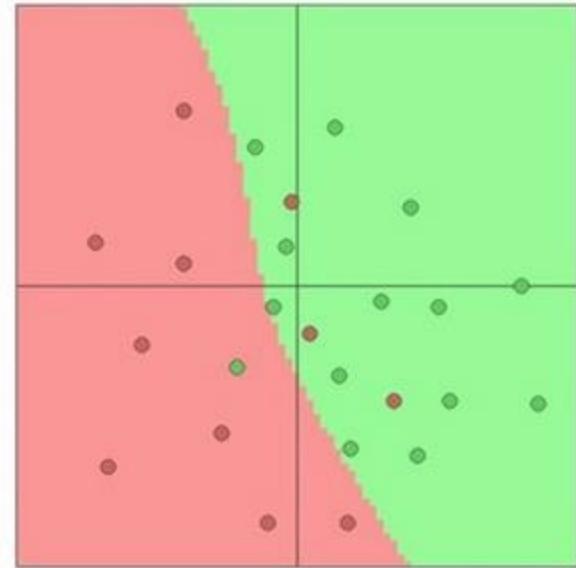
$\lambda = 0.001$



$\lambda = 0.01$



$\lambda = 0.1$



(Web demo with ConvNetJS:

<http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html>)

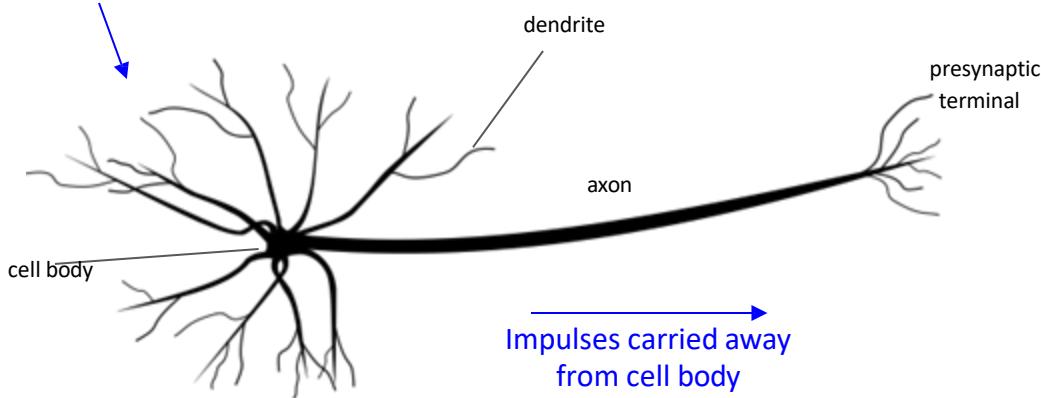
TensorFlow Play Ground: <https://playground.tensorflow.org/>

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



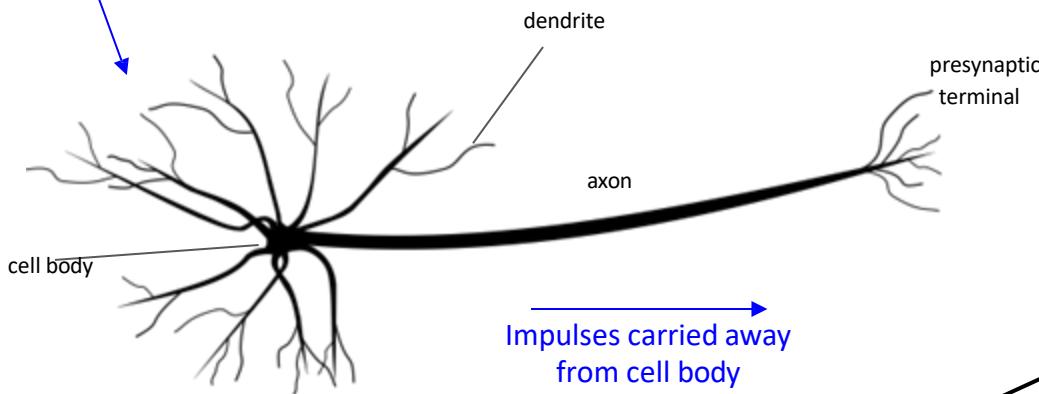
This image by [Fotis Bobolas](#) is licensed under [CC-BY 2.0](#)

Impulses carried toward cell body



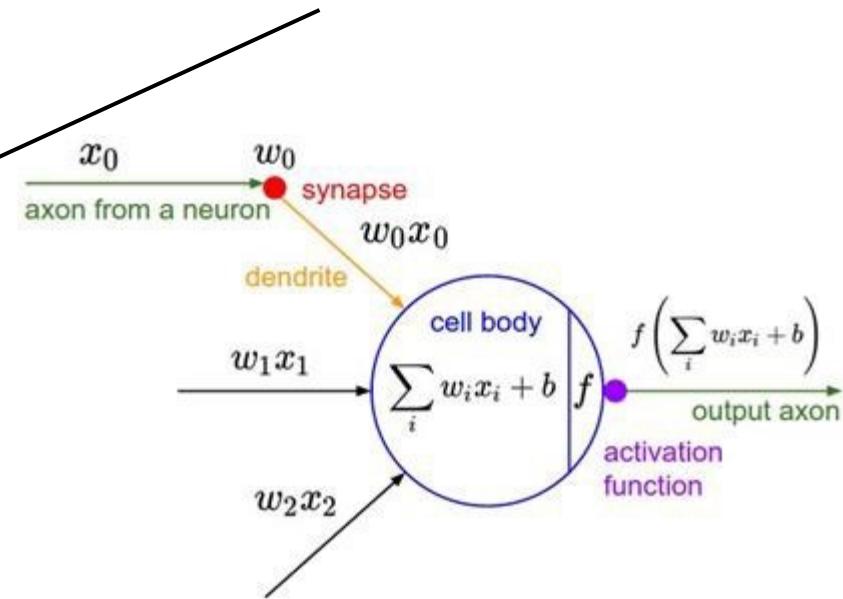
[This image](#) by Felipe PerUCHO  
is licensed under [CC-BY 3.0](#)

Impulses carried toward cell body

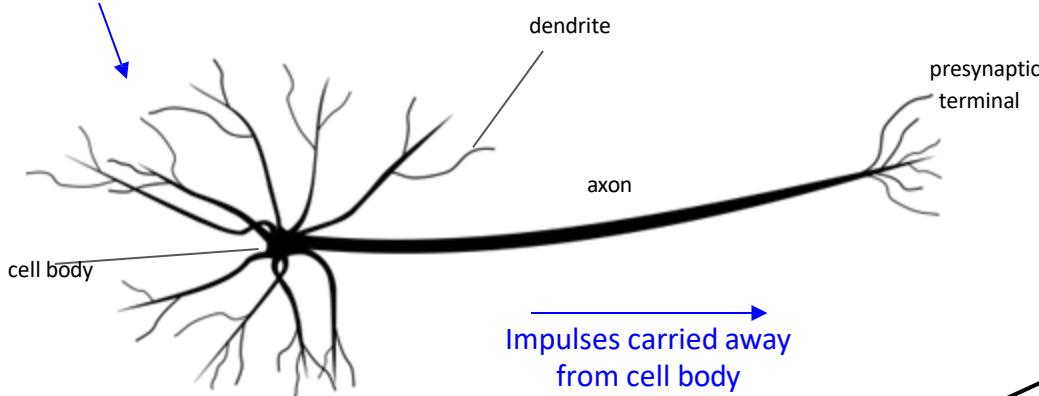


Impulses carried away  
from cell body

This image by Felipe Pericho  
is licensed under CC-BY 3.0

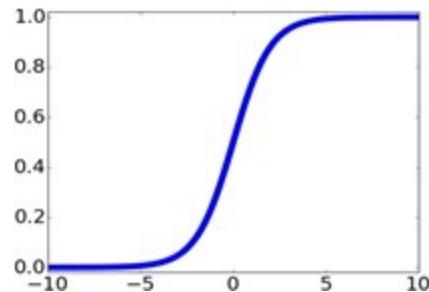


Impulses carried toward cell body



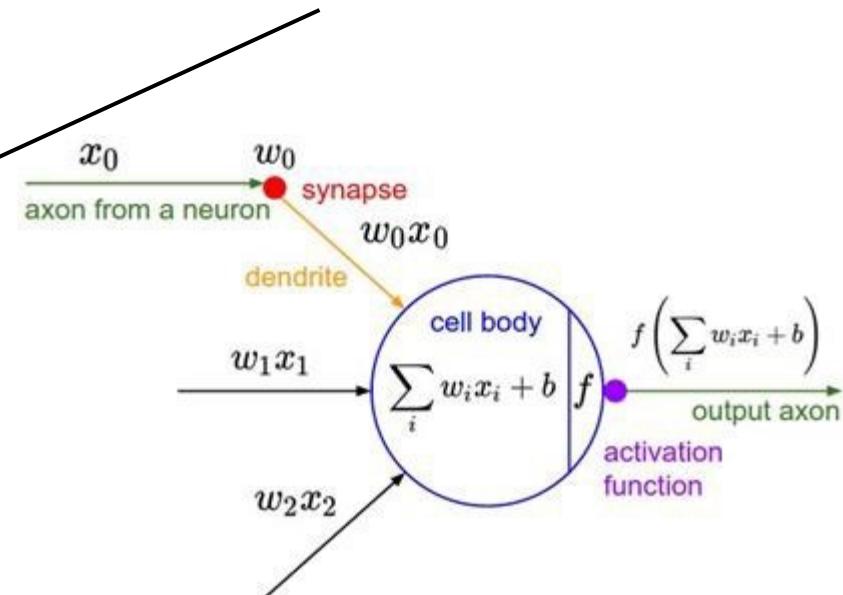
Impulses carried away  
from cell body

This image by Felipe Perucco  
is licensed under CC-BY 3.0

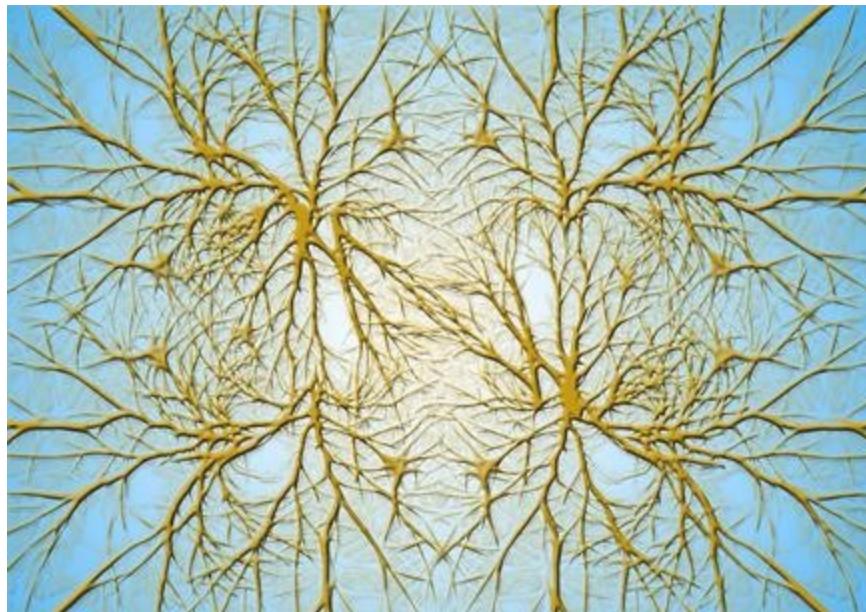


sigmoid activation function

$$\frac{1}{1 + e^{-x}}$$

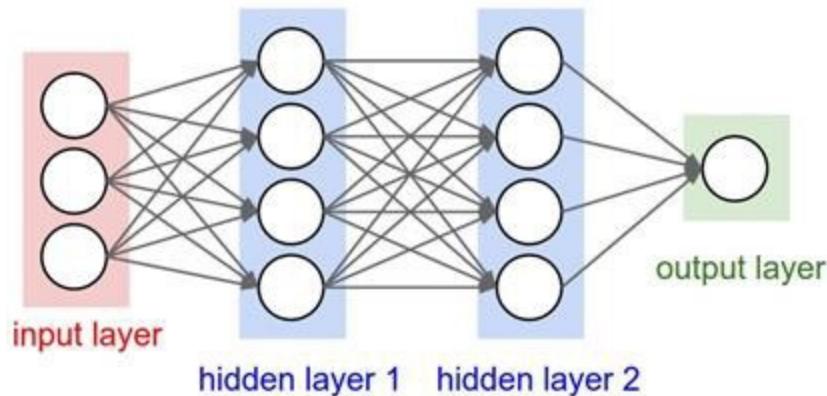


Biological Neurons:  
Complex connectivity patterns

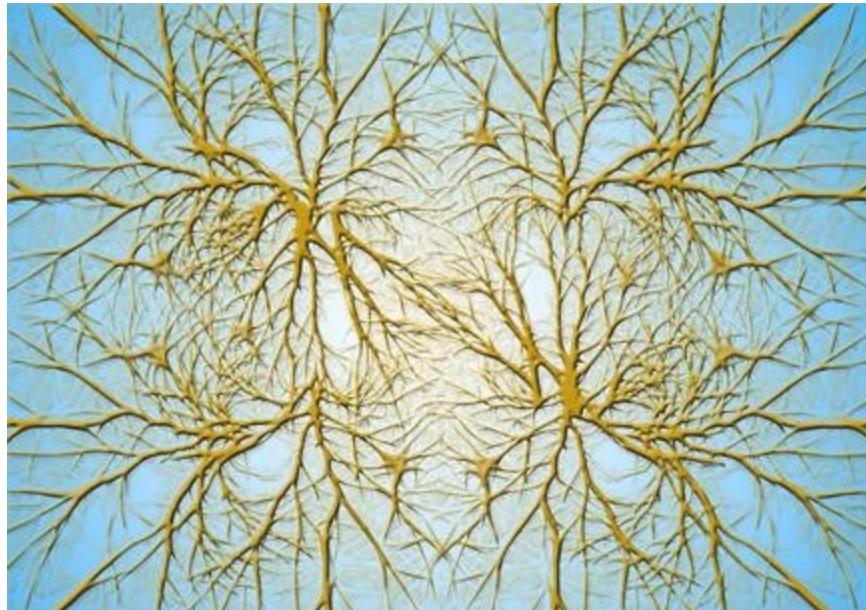


[This image is CC0 Public Domain](#)

Neurons in a neural network:  
Organized into regular layers for  
computational efficiency

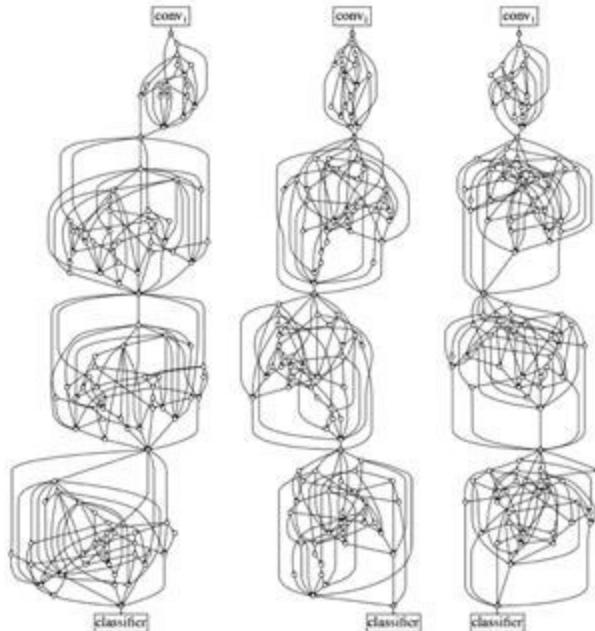


# Biological Neurons: Complex connectivity patterns



[This image is CC0 Public Domain](#)

But neural networks with random connections can work too!



Xie et al, "Exploring Randomly Wired Neural Networks for Image Recognition", IEEE/CVF International Conference on Computer Vision 2019

# Be very careful with your brain analogies!

Biological Neurons:

- Many different types
- Dendrites can perform complex non-linear computations
- Synapses are not a single weight but a complex non-linear dynamical system

[Dendritic Computation. London and Häusser]

# Plugging in neural networks with loss functions

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{Hinge Loss on predictions}$$

$$R(W) = \sum_k W_k^2 \quad \text{Regularization}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization}$$

# Problem: How to compute gradients?

$$s = f(x; W_1, W_2) = W_2 \max(0, W_1 x) \quad \text{Nonlinear score function}$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1) \quad \text{Hinge Loss on predictions}$$

$$R(W) = \sum_k W_k^2 \quad \text{Regularization}$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda R(W_1) + \lambda R(W_2) \quad \text{Total loss: data loss + regularization}$$

If we can compute  $\frac{\partial L}{\partial W_1}, \frac{\partial L}{\partial W_2}$  then we can learn  $W_1$  and  $W_2$

# (Bad) Idea: Derive $\nabla_W L$ on paper

$$s = f(x; W) = Wx$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1)$$

$$L = \frac{1}{N} \sum_{i=1}^N L_i + \lambda \sum_k W_k^2$$

$$= \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2$$

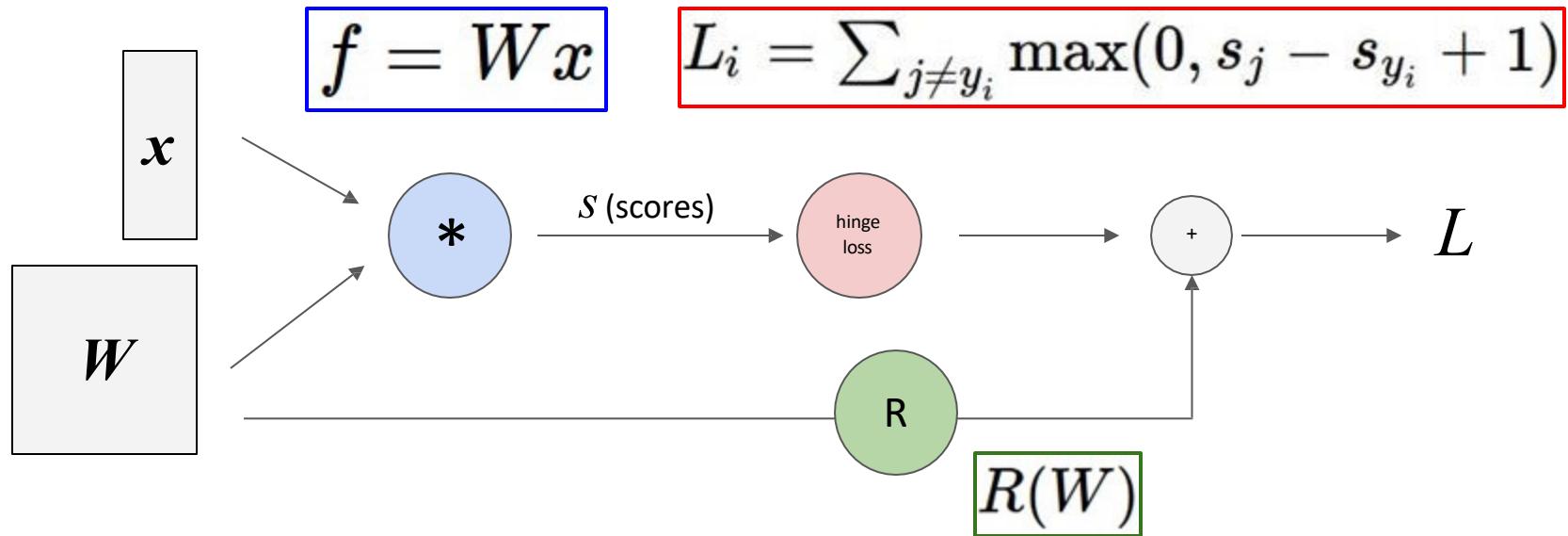
$$\nabla_W L = \nabla_W \left( \frac{1}{N} \sum_{i=1}^N \sum_{j \neq y_i} \max(0, W_{j,:} \cdot x + W_{y_i,:} \cdot x + 1) + \lambda \sum_k W_k^2 \right)$$

Problem: Very tedious: Lots of matrix calculus, need lots of paper

Problem: What if we want to change loss? E.g. use softmax instead of hinge? Need to re-derive everything from scratch!

Problem: Not feasible for very complex models!

# Better Idea: Computational graphs + Backpropagation



# Convolutional network (AlexNet)

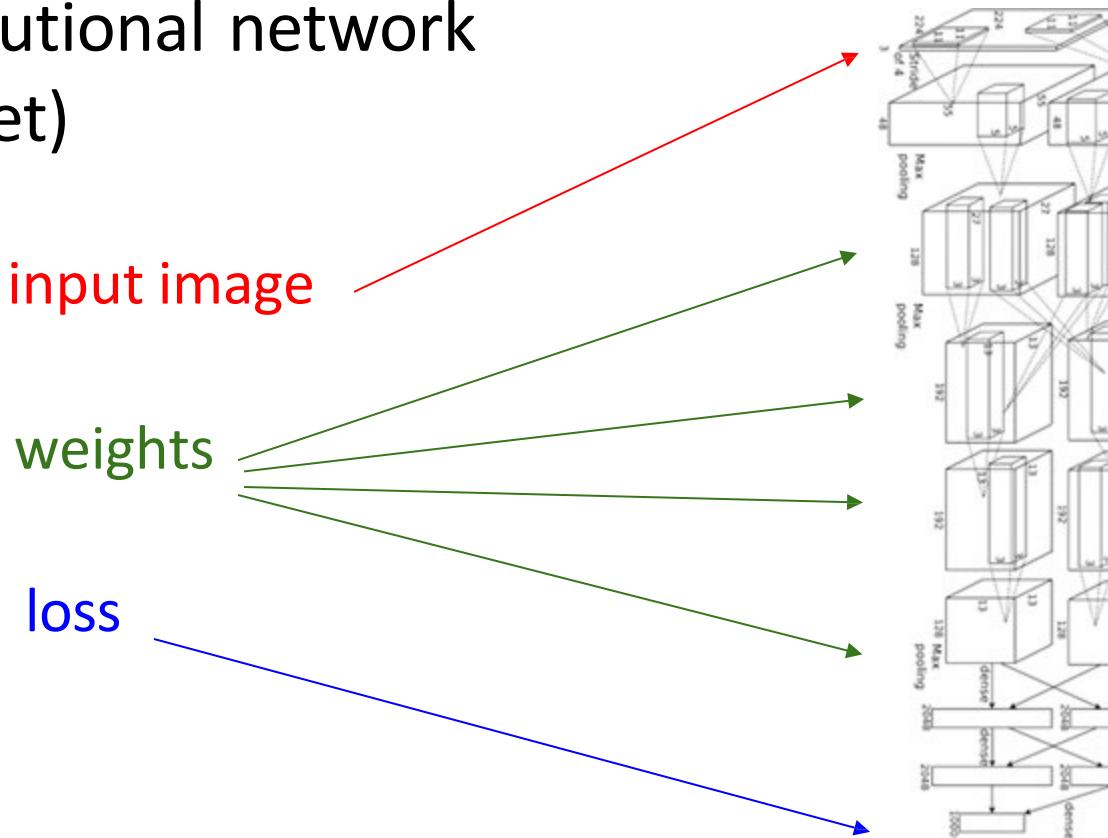


Figure copyright Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton, 2012. Reproduced with permission.

# Really complex neural networks!!

input image

loss

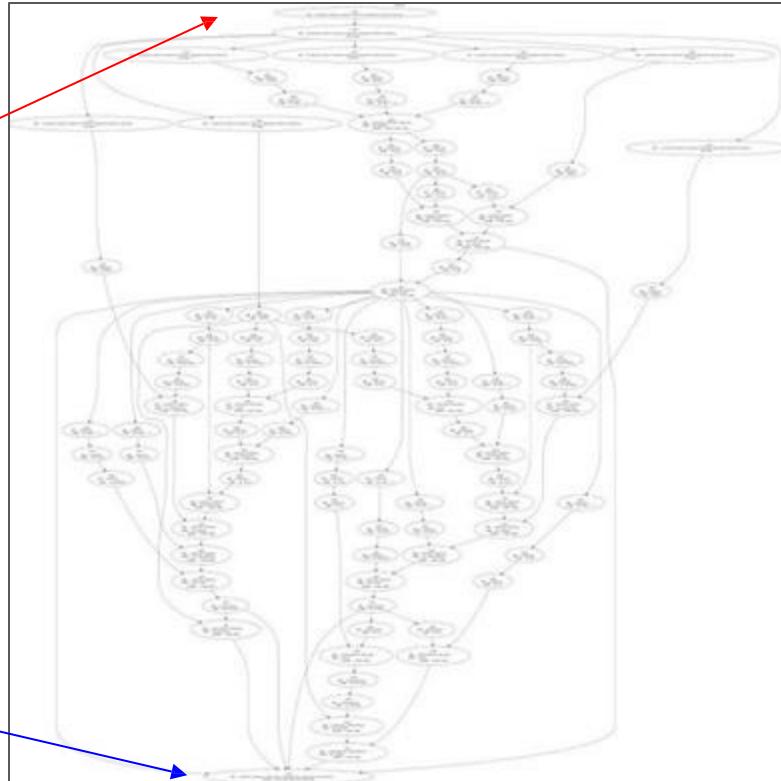


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

# Neural Turing Machine

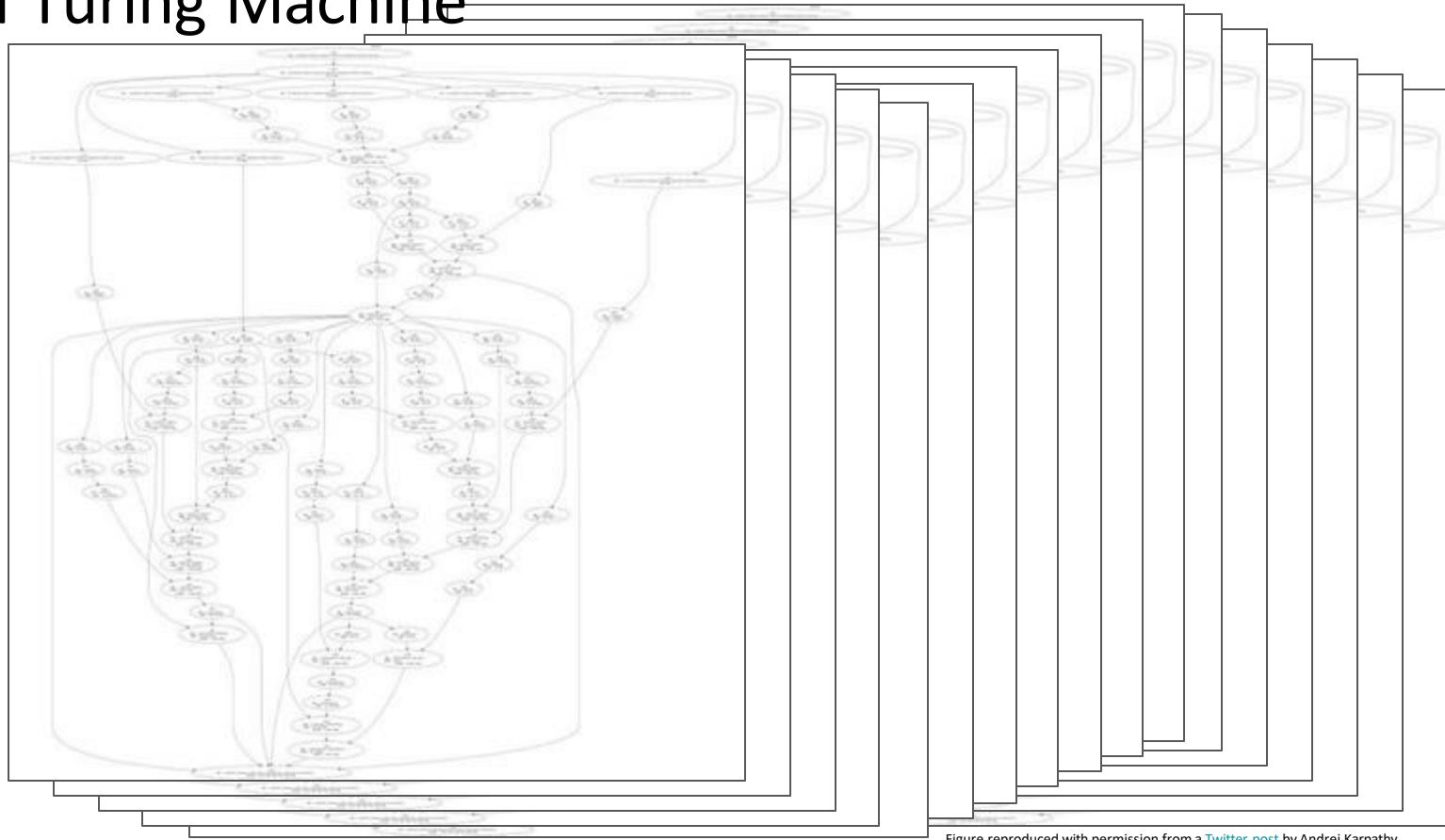


Figure reproduced with permission from a [Twitter post](#) by Andrej Karpathy.

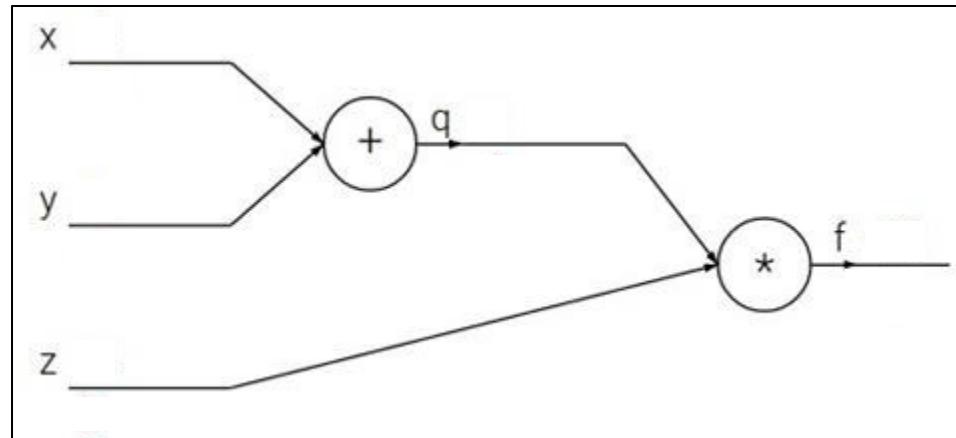
# Solution: Backpropagation

## Backpropagation: a simple example

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

## Backpropagation: a simple example

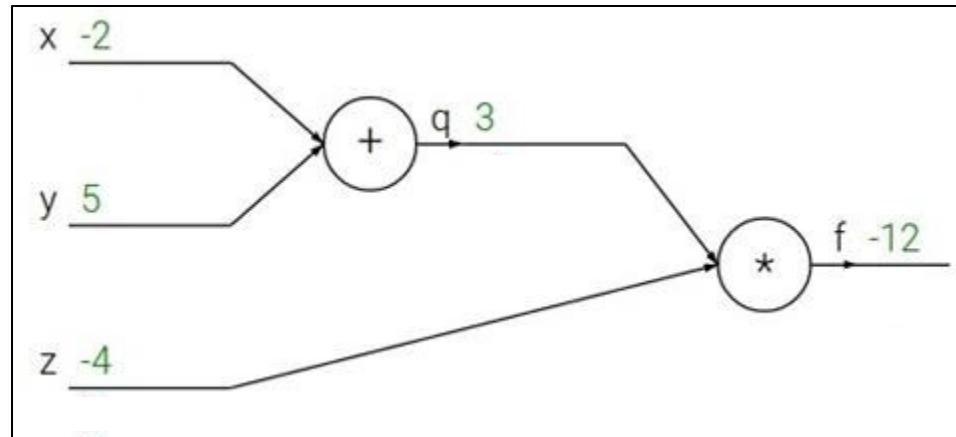
$$f(x, y, z) = (x + y)z$$



## Backpropagation: a simple example

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

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



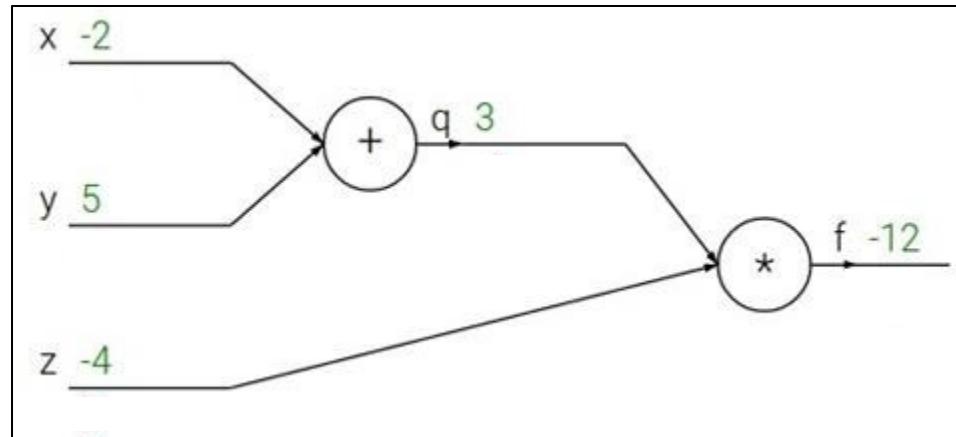
## Backpropagation: a simple example

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

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

4

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



## Backpropagation: a simple example

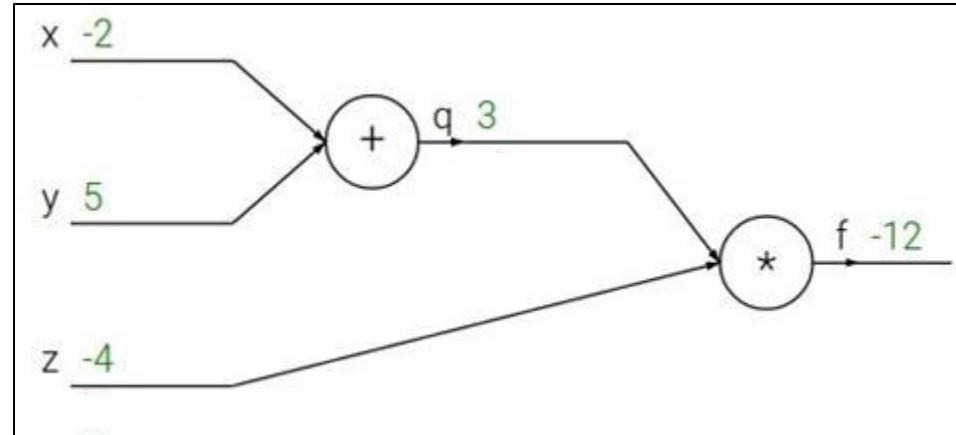
$$f(x, y, z) = (x + y)z$$

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

4

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

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



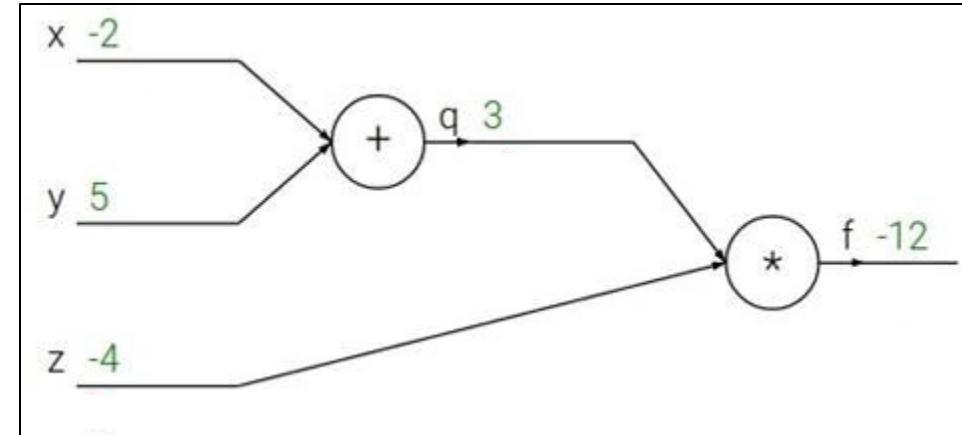
## Backpropagation: a simple example

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

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

4

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



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

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

## Backpropagation: a simple example

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

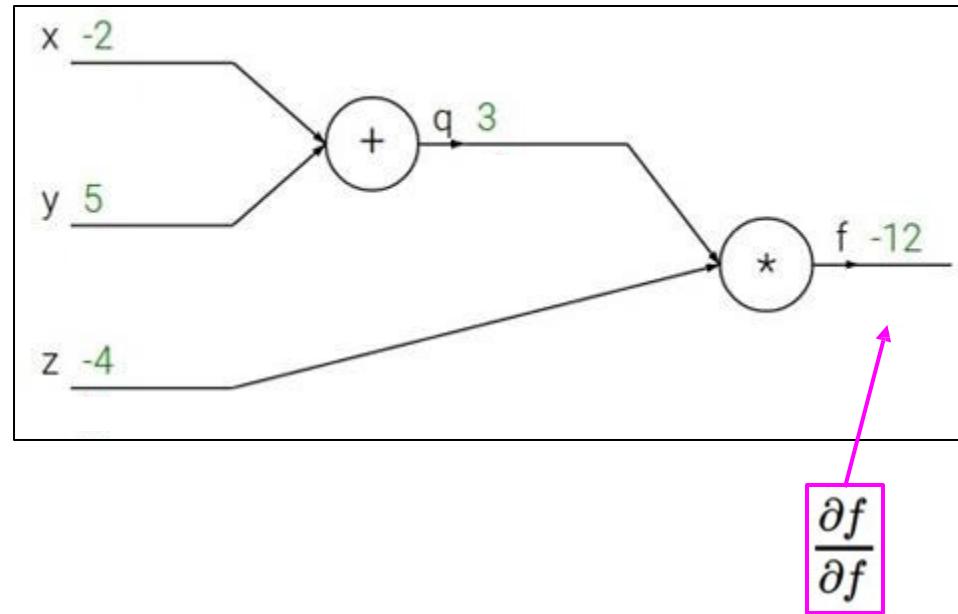
e.g.  $x = -2, y = 5, z = -$

4

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

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

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



## Backpropagation: a simple example

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

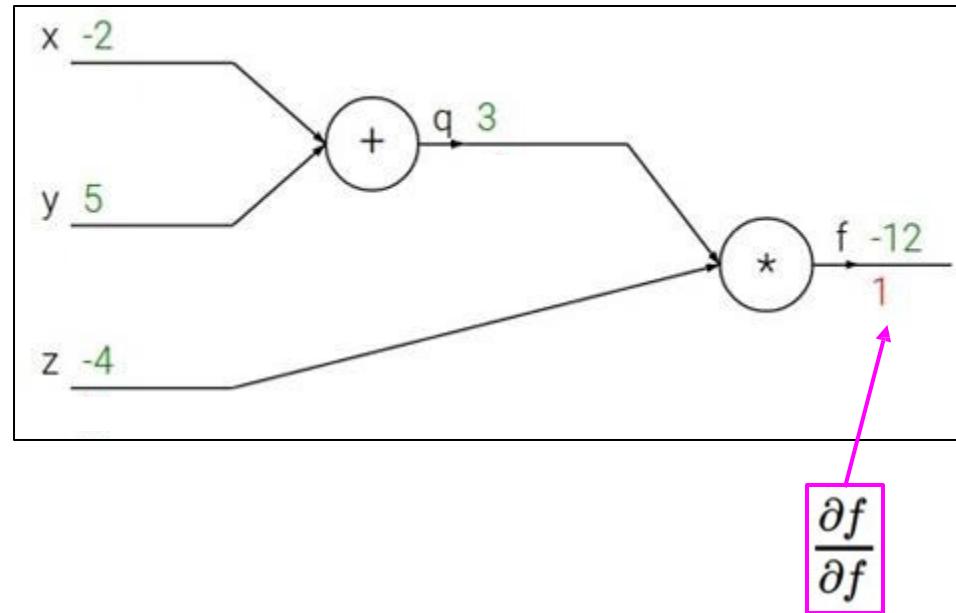
e.g.  $x = -2, y = 5, z = -$

4

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

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

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



## Backpropagation: a simple example

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

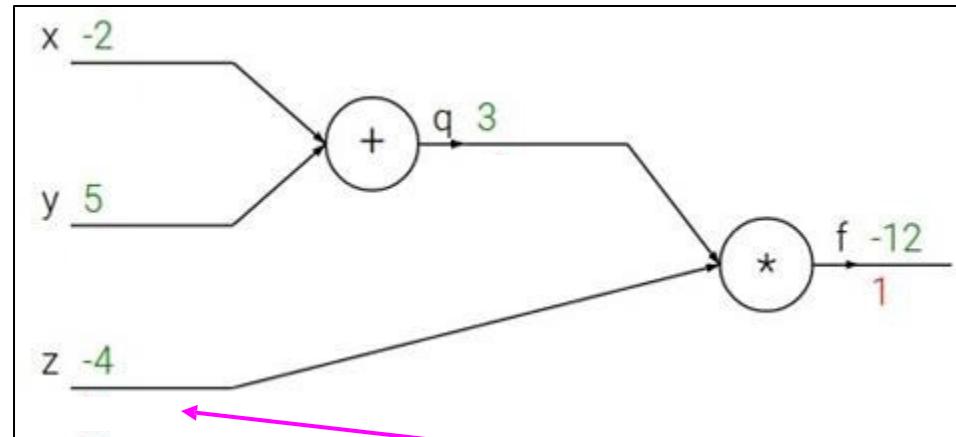
e.g.  $x = -2, y = 5, z = -$

4

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

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

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



$$\frac{\partial f}{\partial z}$$

## Backpropagation: a simple example

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

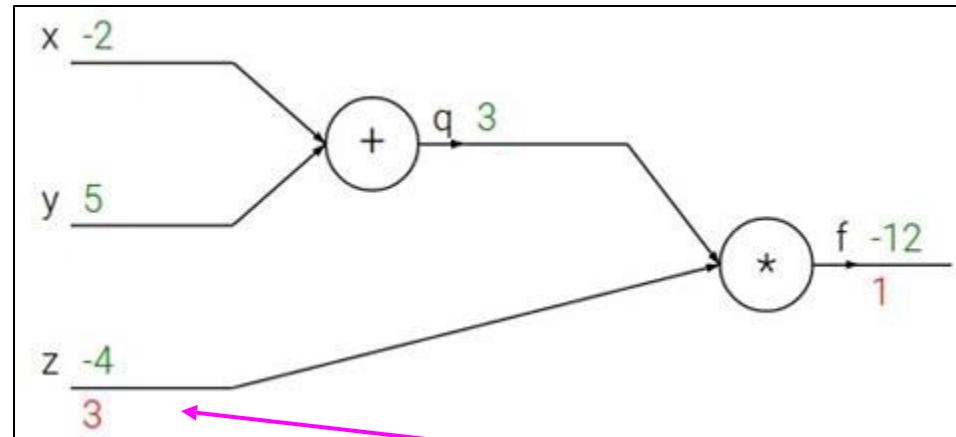
e.g.  $x = -2, y = 5, z = -$

4

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

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

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



$$\frac{\partial f}{\partial z}$$

## Backpropagation: a simple example

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

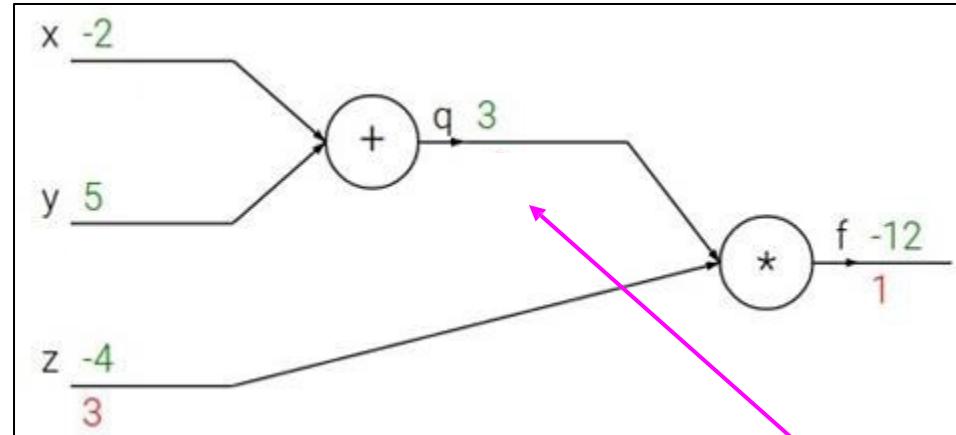
e.g.  $x = -2, y = 5, z = -$

4

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

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

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



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

## Backpropagation: a simple example

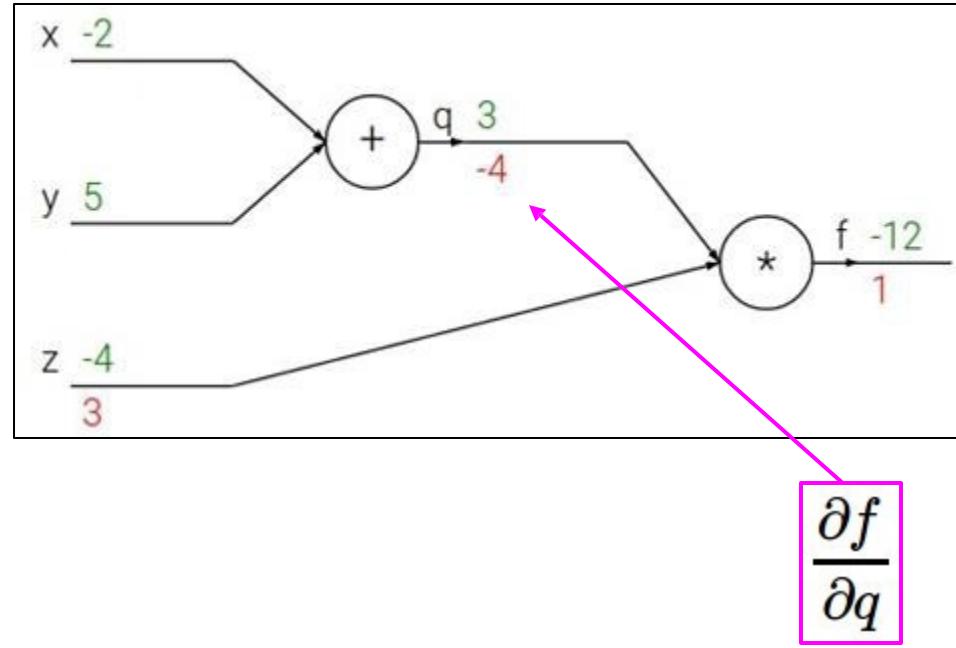
$$f(x, y, z) = (x + y)z$$

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

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

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

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



## Backpropagation: a simple example

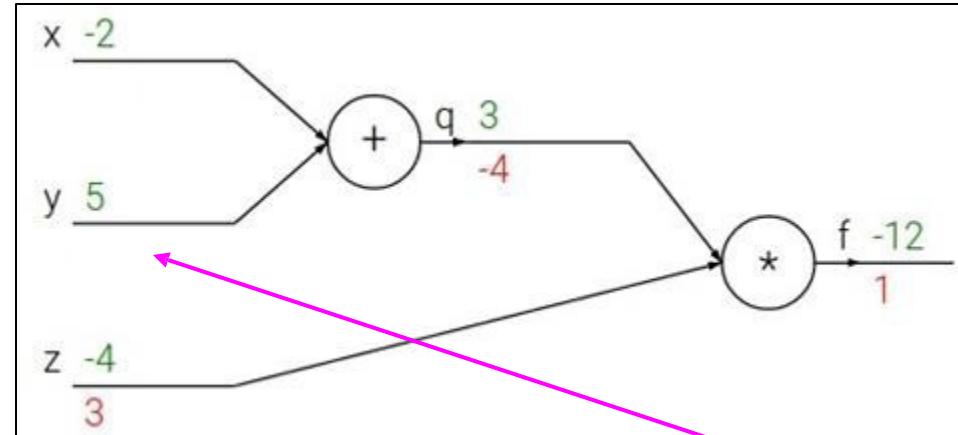
$$f(x, y, z) = (x + y)z$$

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

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

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

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



Chain rule:

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

Upstream  
gradient

Local  
gradient

## Backpropagation: a simple example

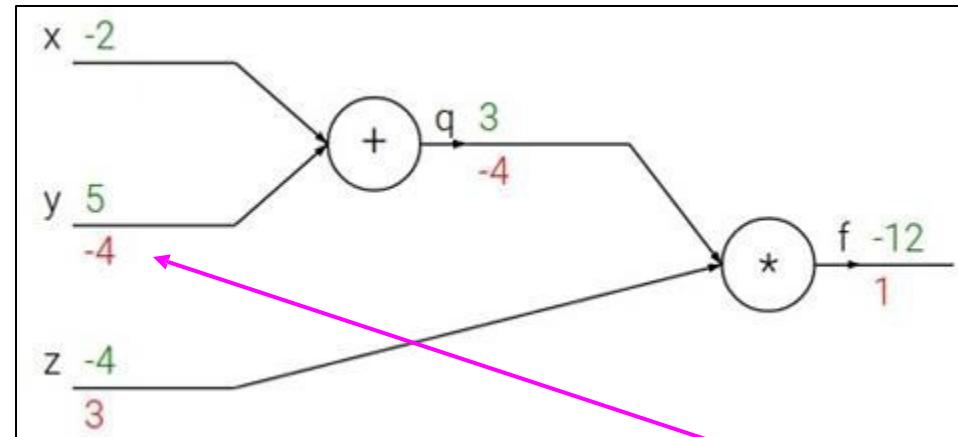
$$f(x, y, z) = (x + y)z$$

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

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

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

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



Chain rule:

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

Upstream  
gradient

Local  
gradient

## Backpropagation: a simple example

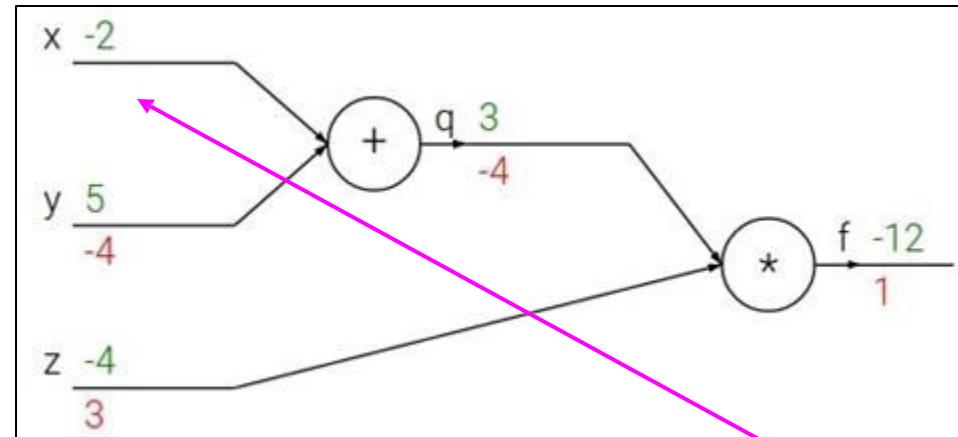
$$f(x, y, z) = (x + y)z$$

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

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

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

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



Chain rule:

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

Upstream  
gradient

Local  
gradient

## Backpropagation: a simple example

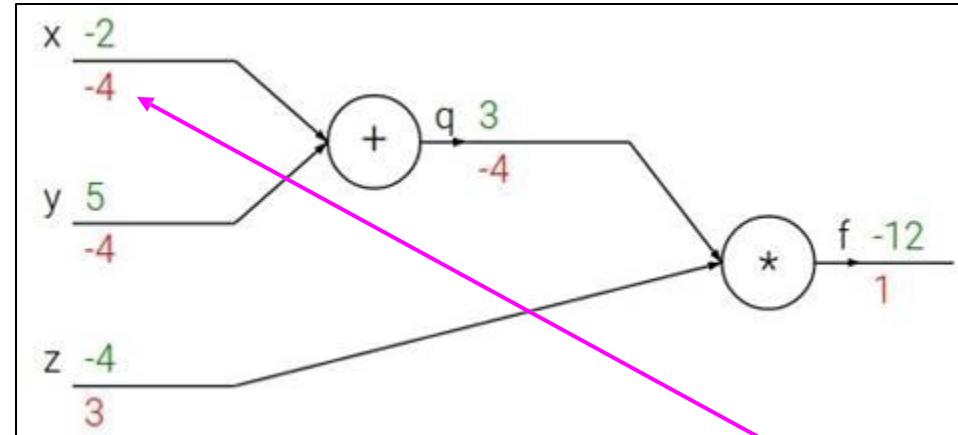
$$f(x, y, z) = (x + y)z$$

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

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

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

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

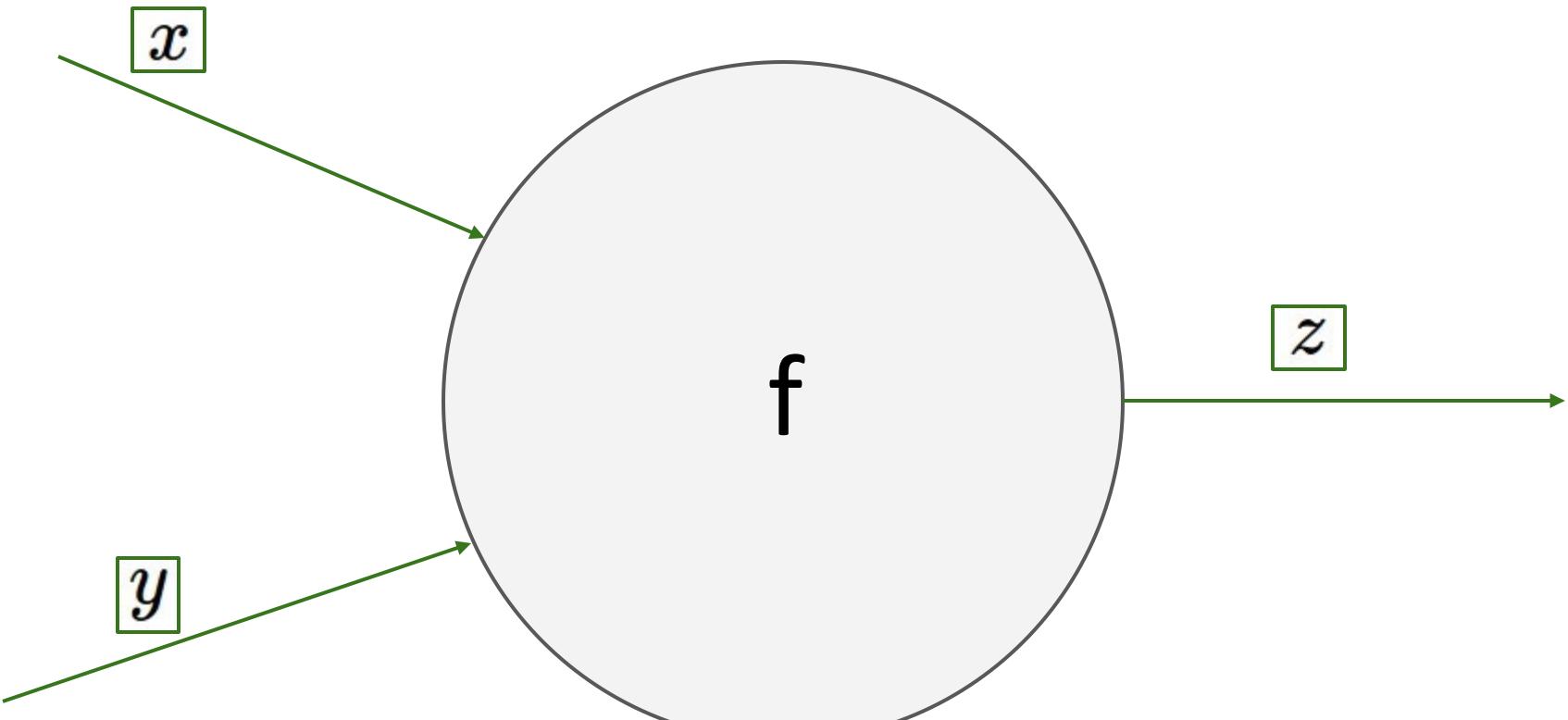


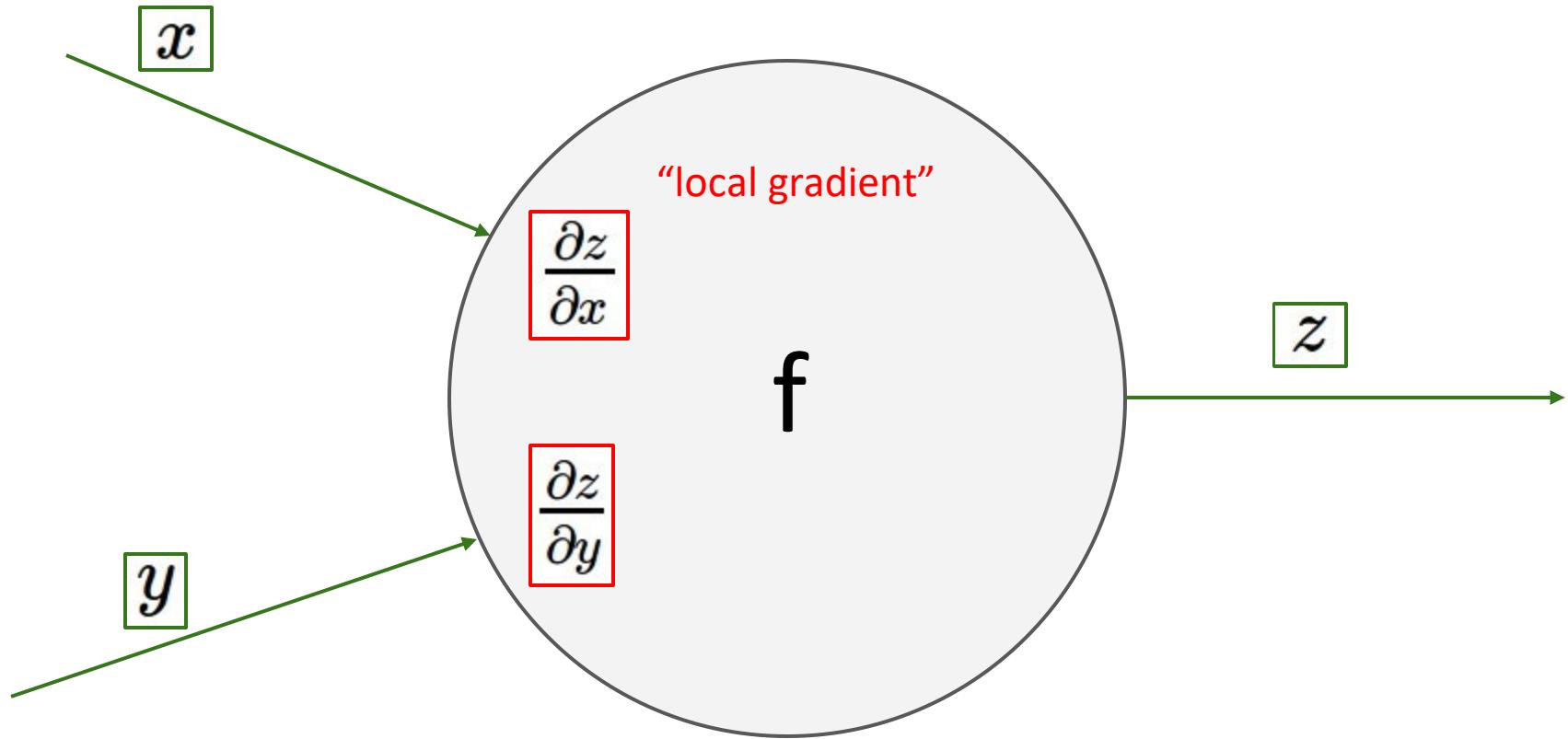
Chain rule:

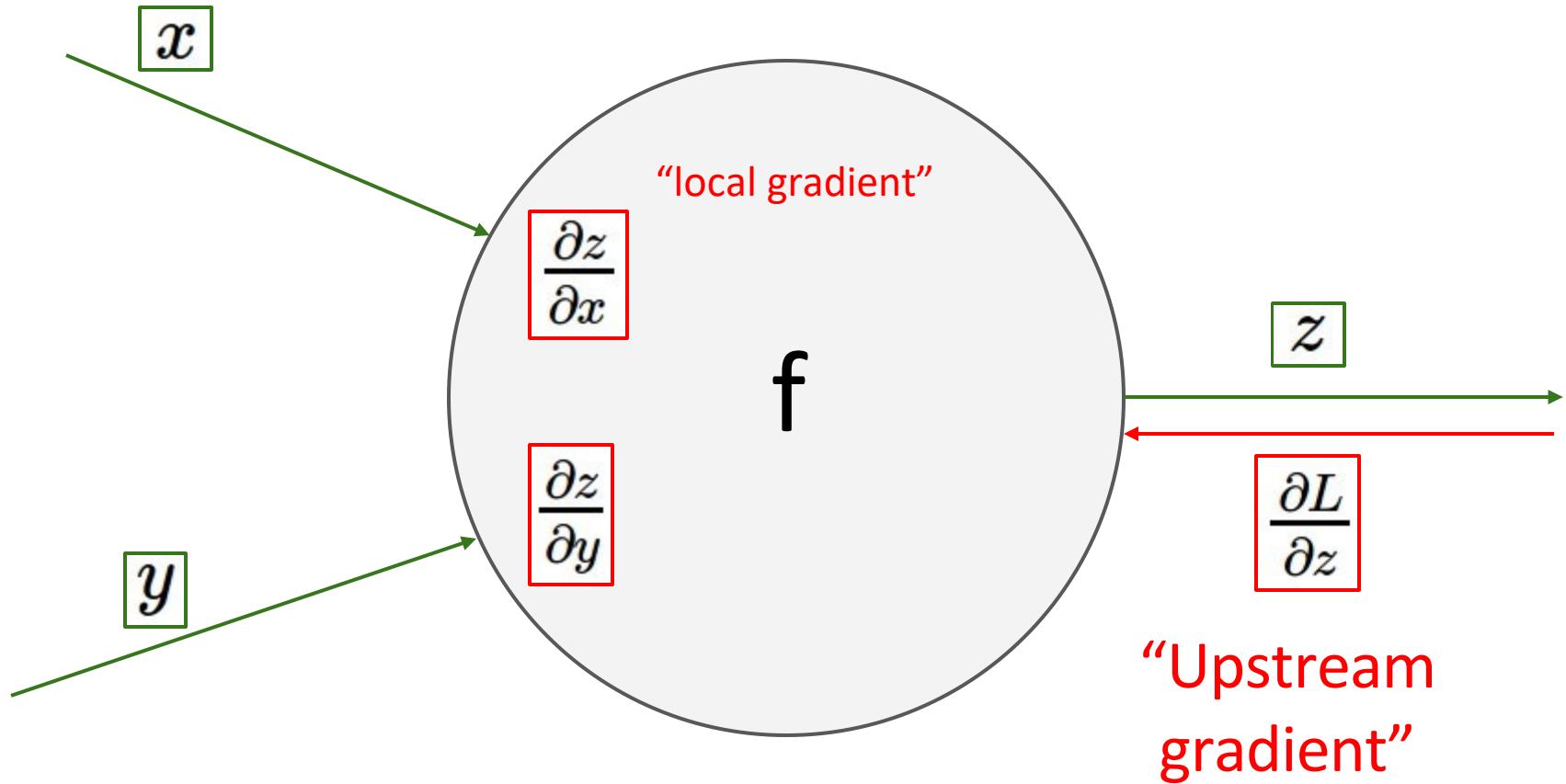
$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x}$$

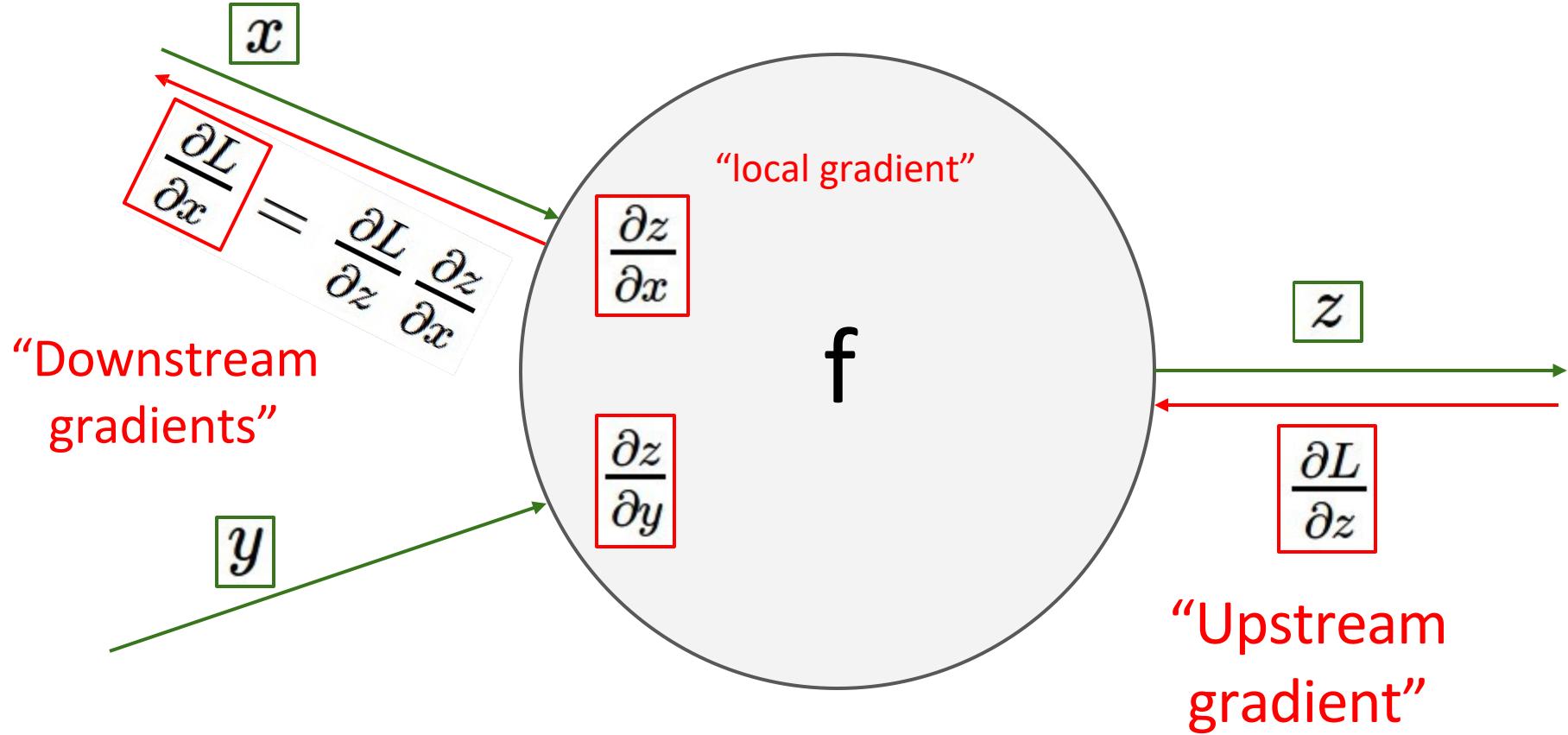
Upstream  
gradient

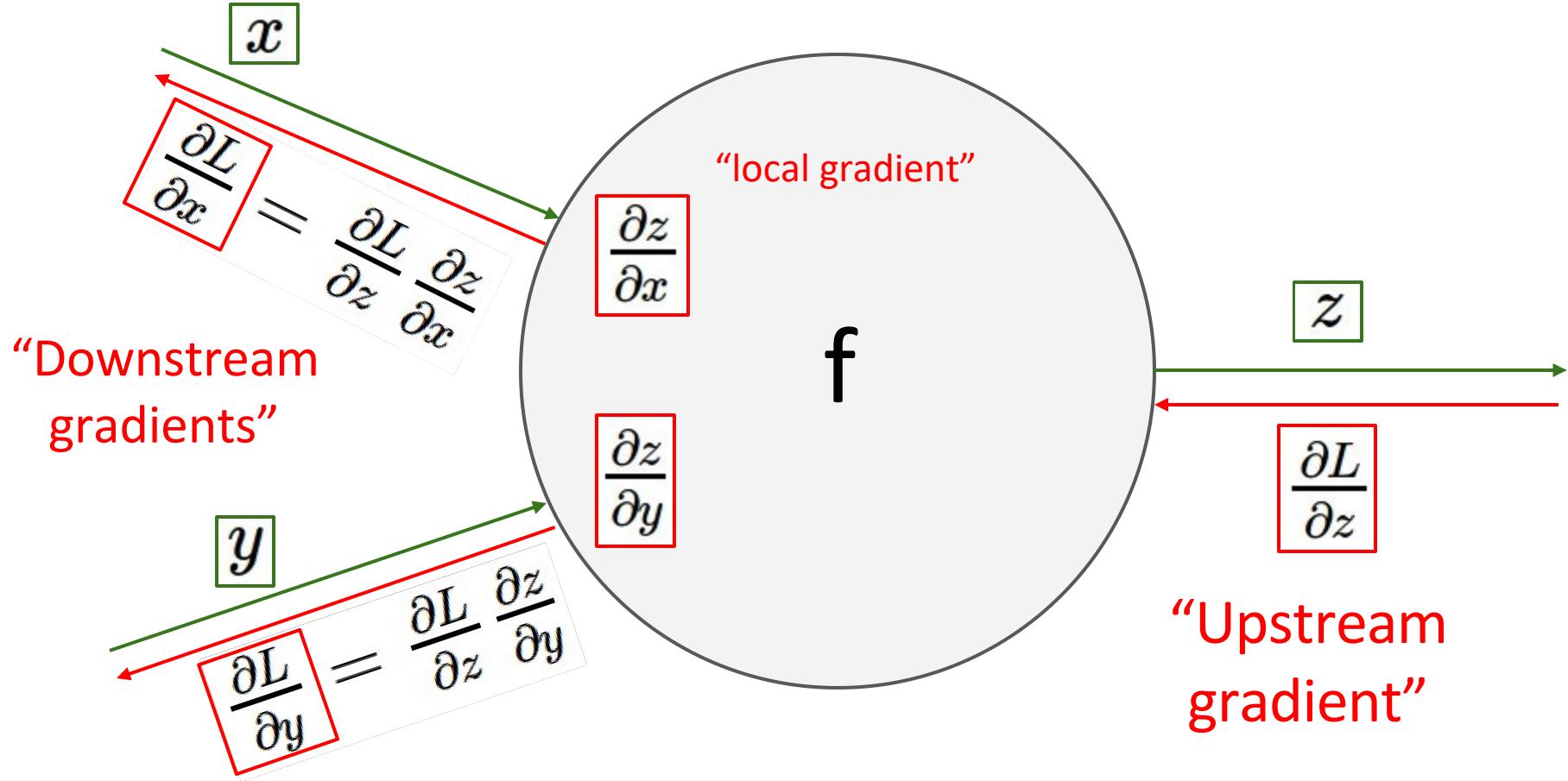
Local  
gradient

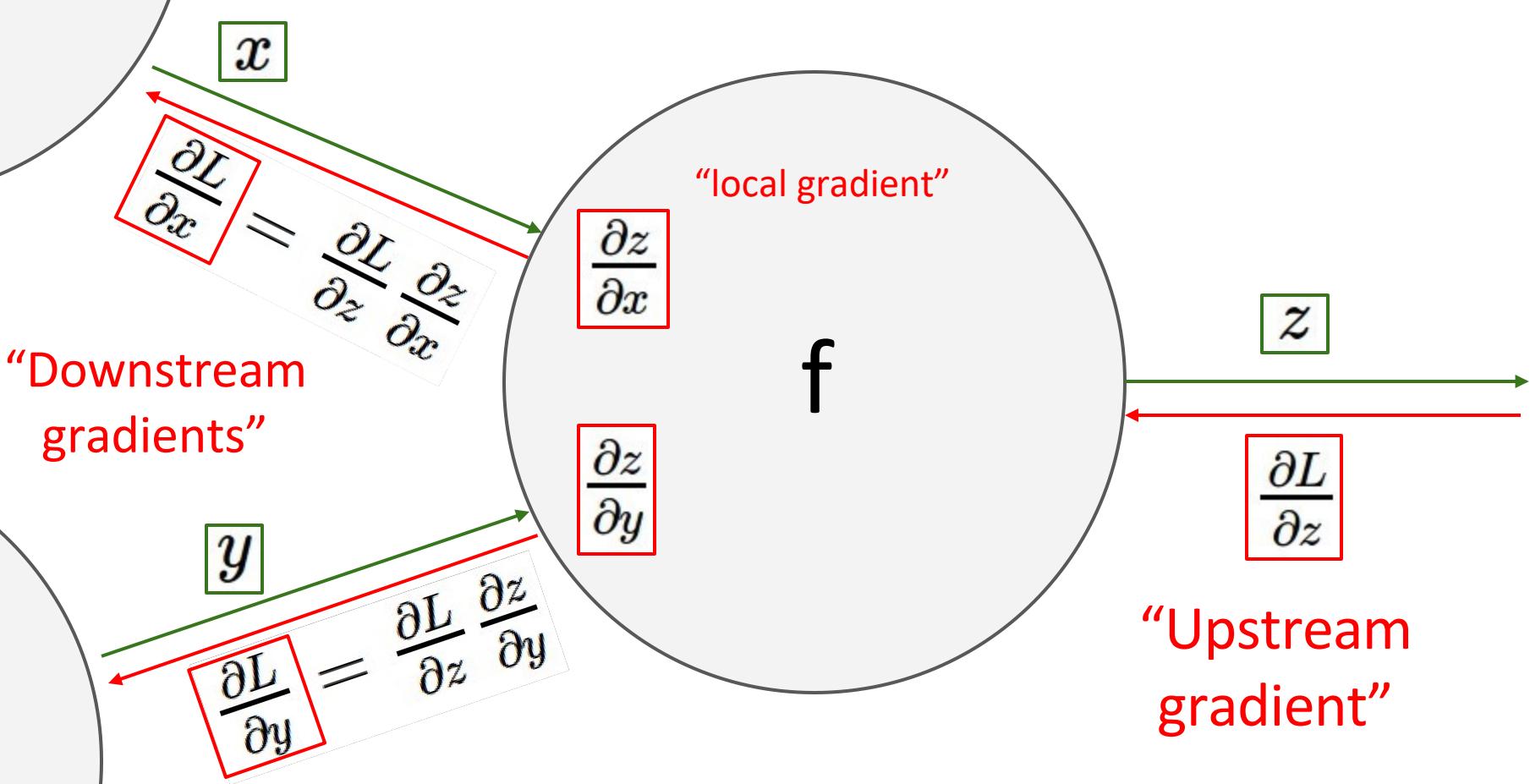






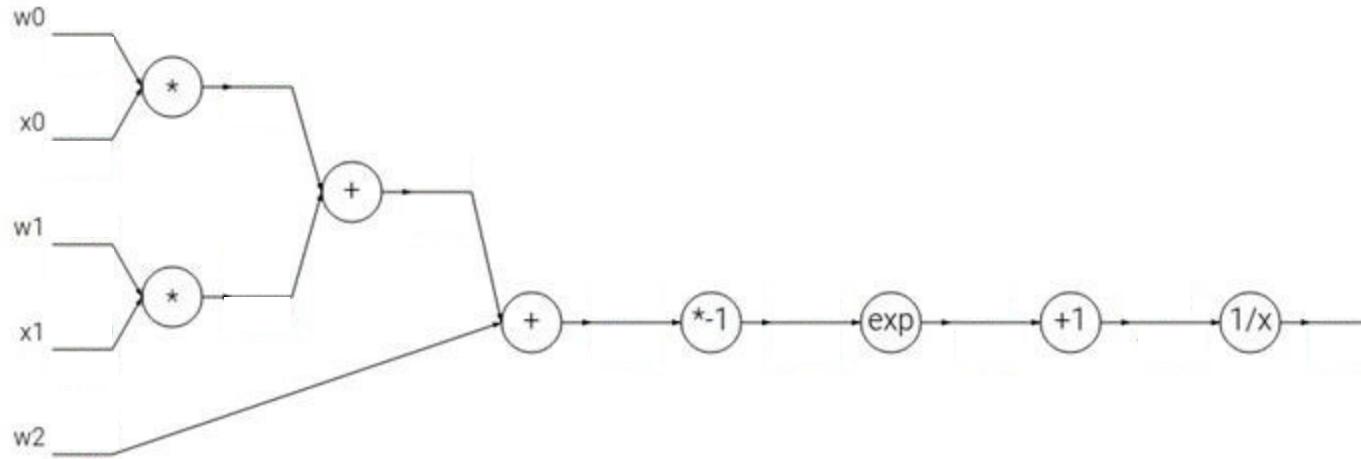






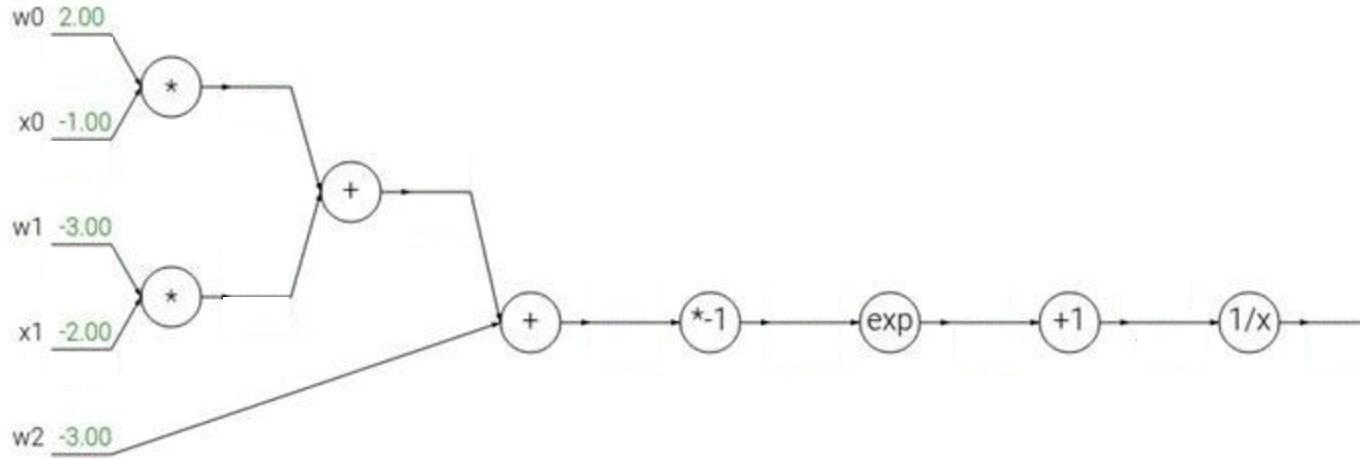
Another example:

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



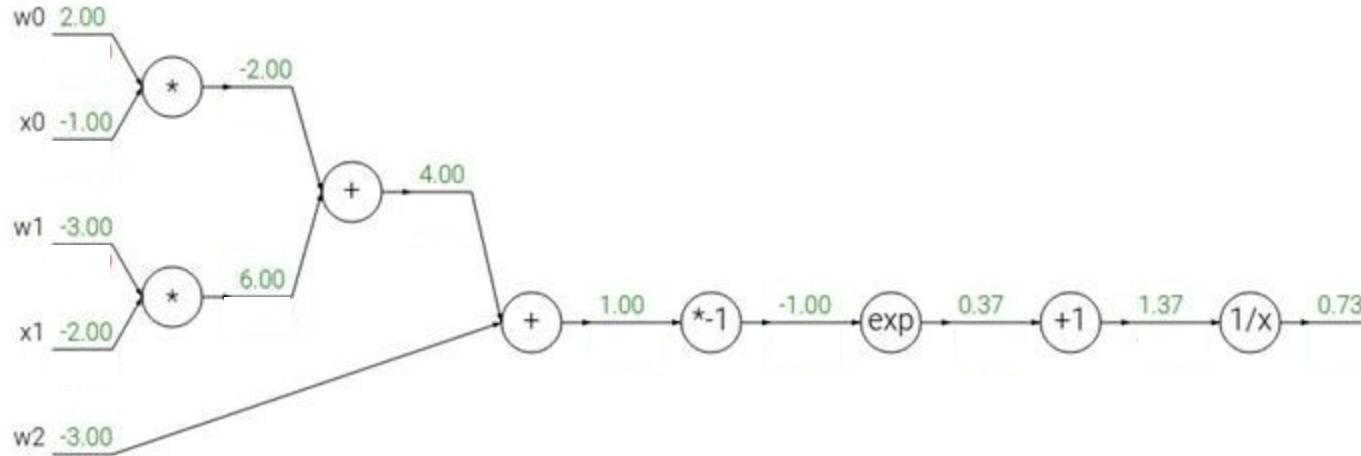
Another example:

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



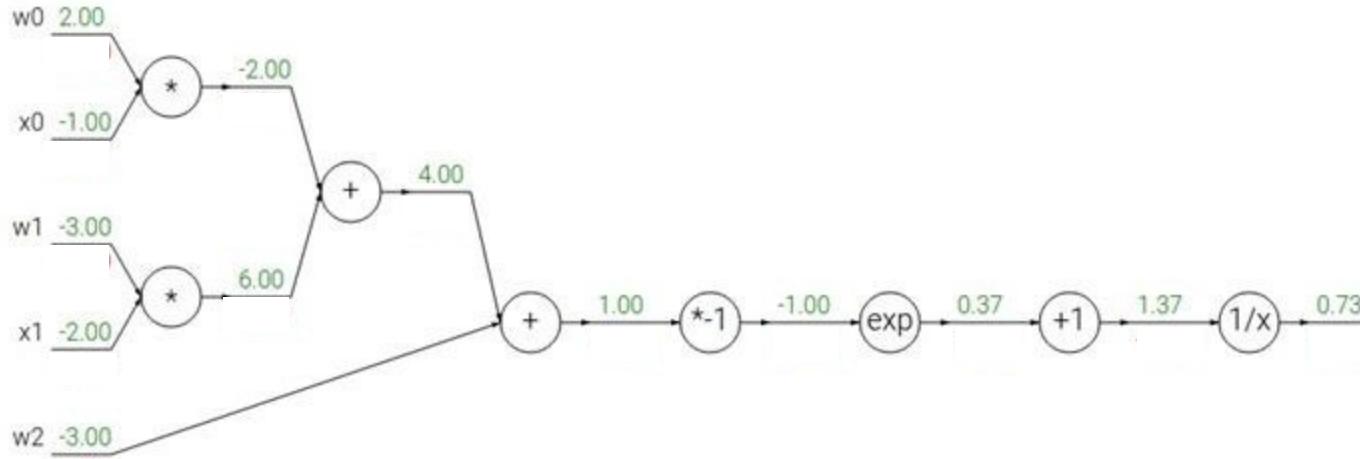
Another example:

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



Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

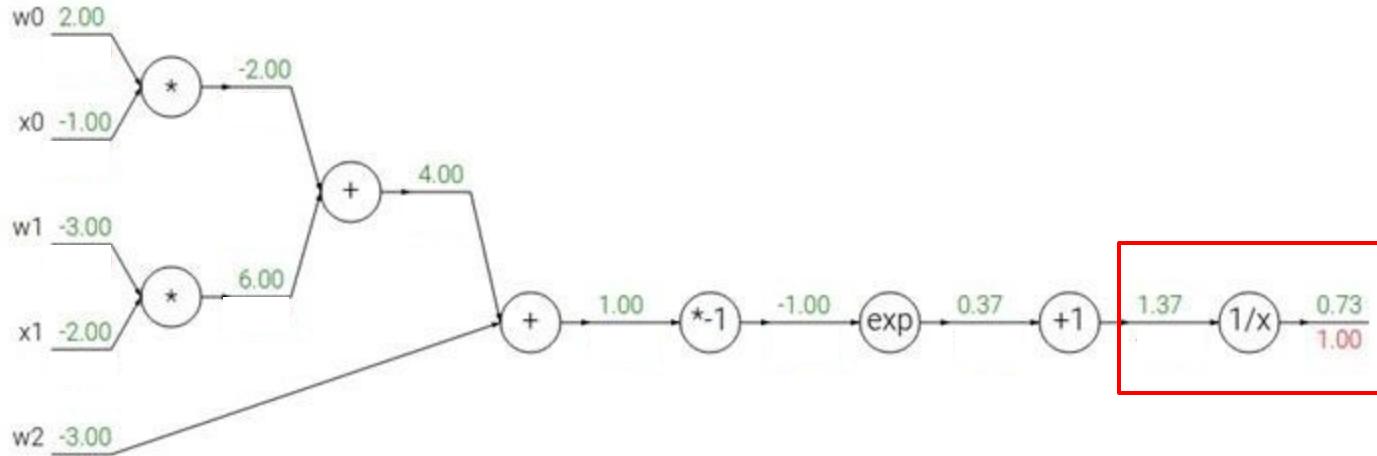
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

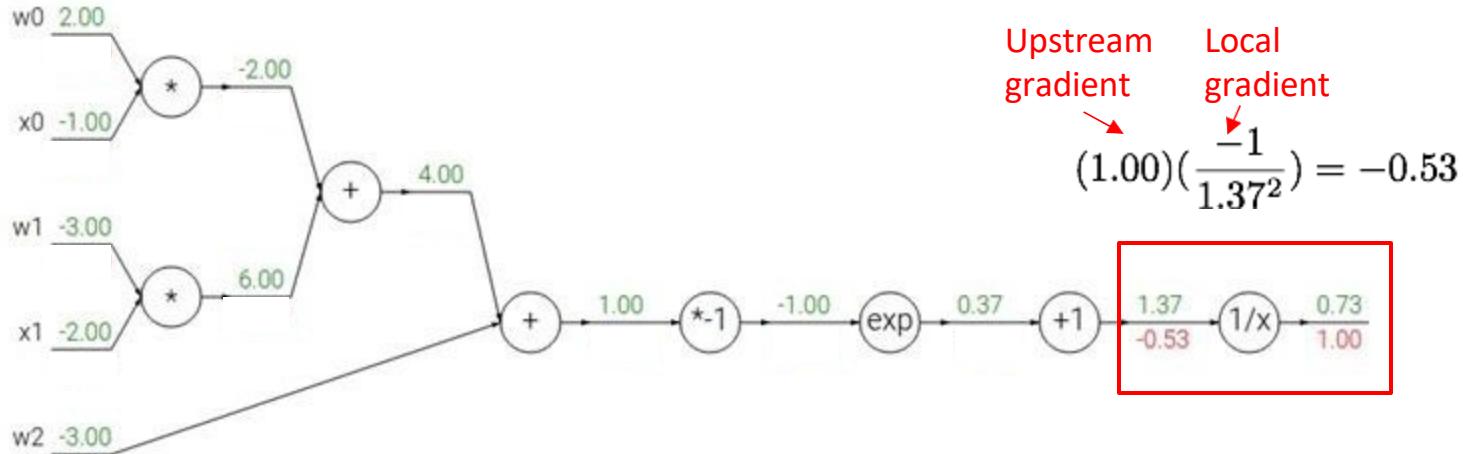
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

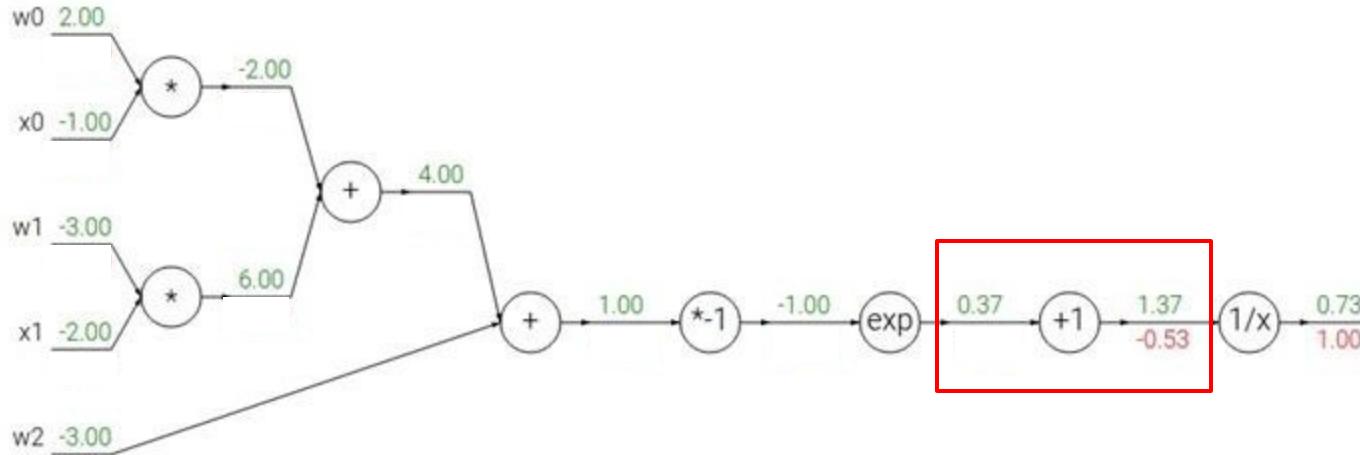
$$f_c(x) = c + x$$

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

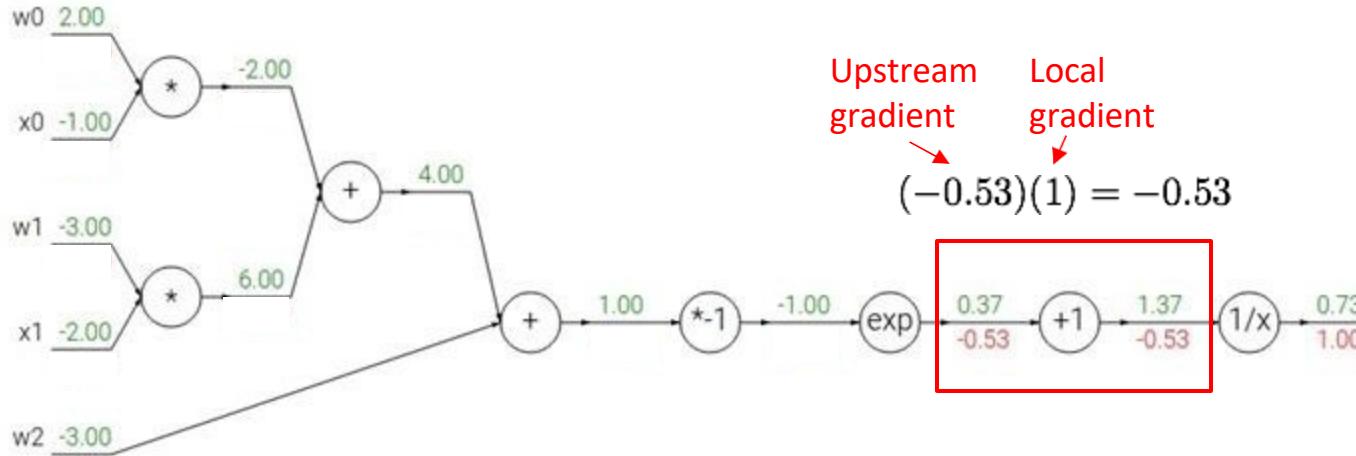
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

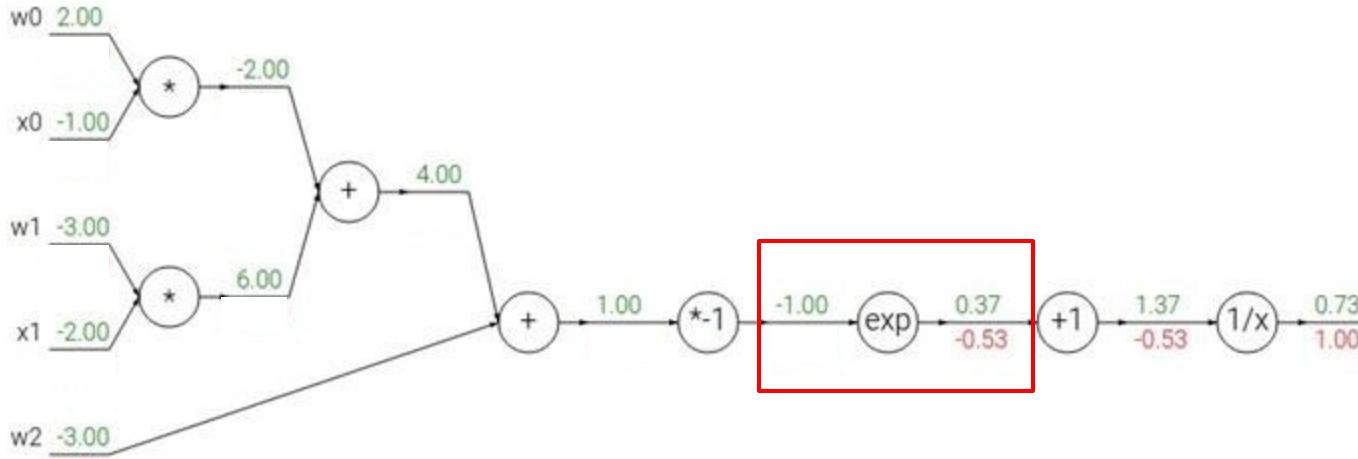
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

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



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

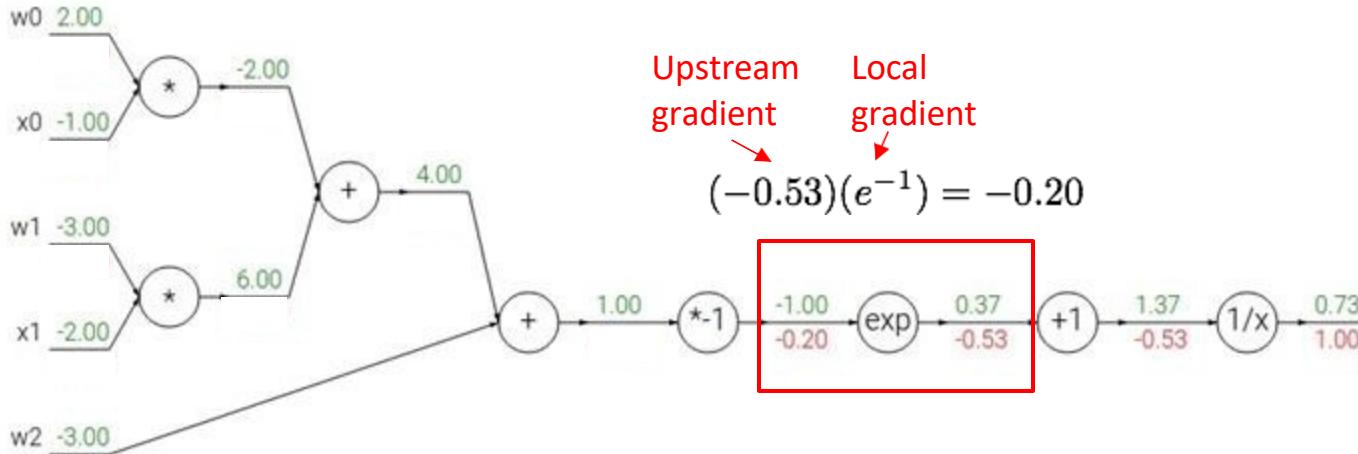
$$f_a(x) = ax \quad \rightarrow \quad \frac{df}{dx} = a$$

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

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

Another example:

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



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

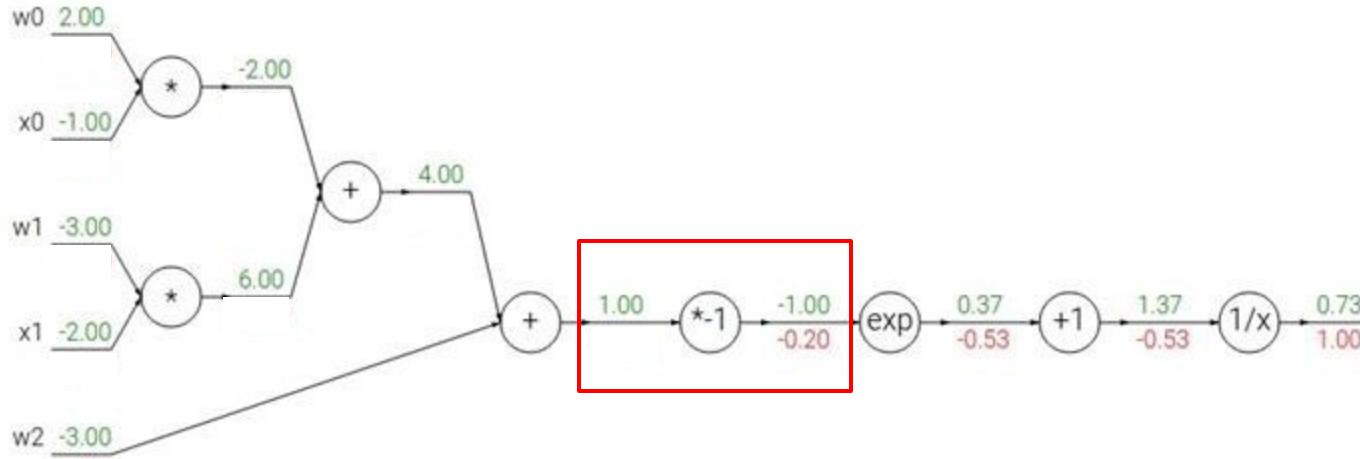
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

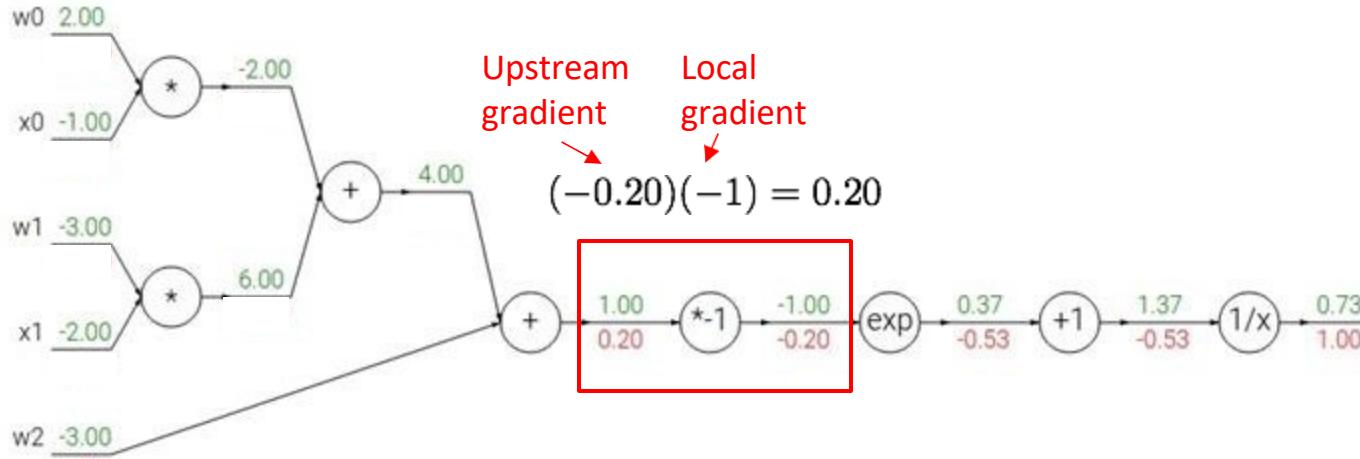
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

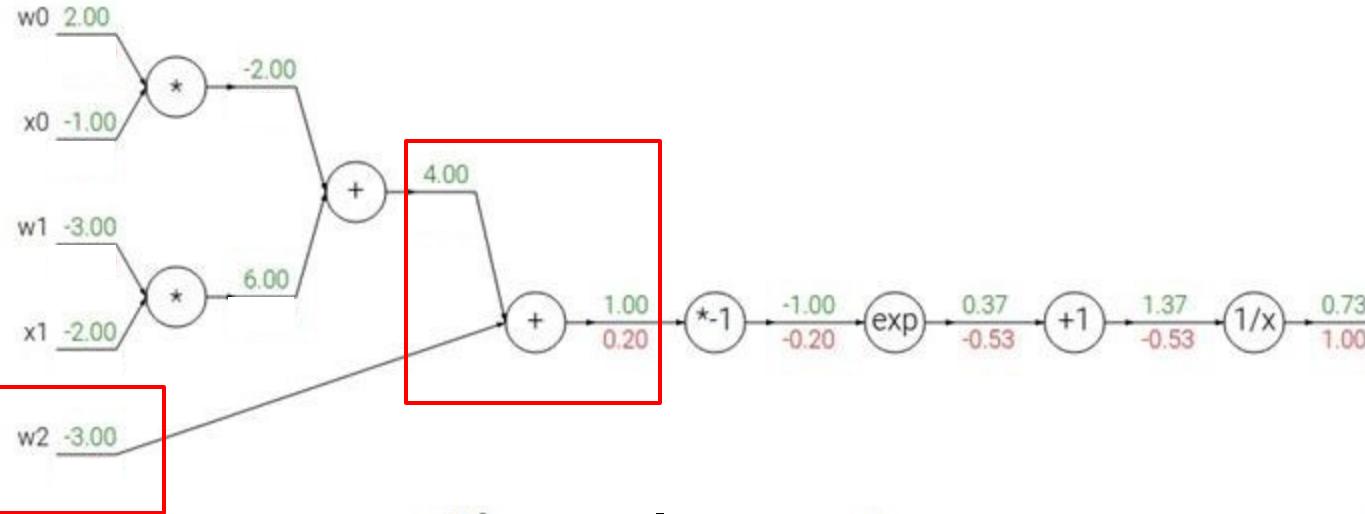
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$$f_c(x) = c + x$$

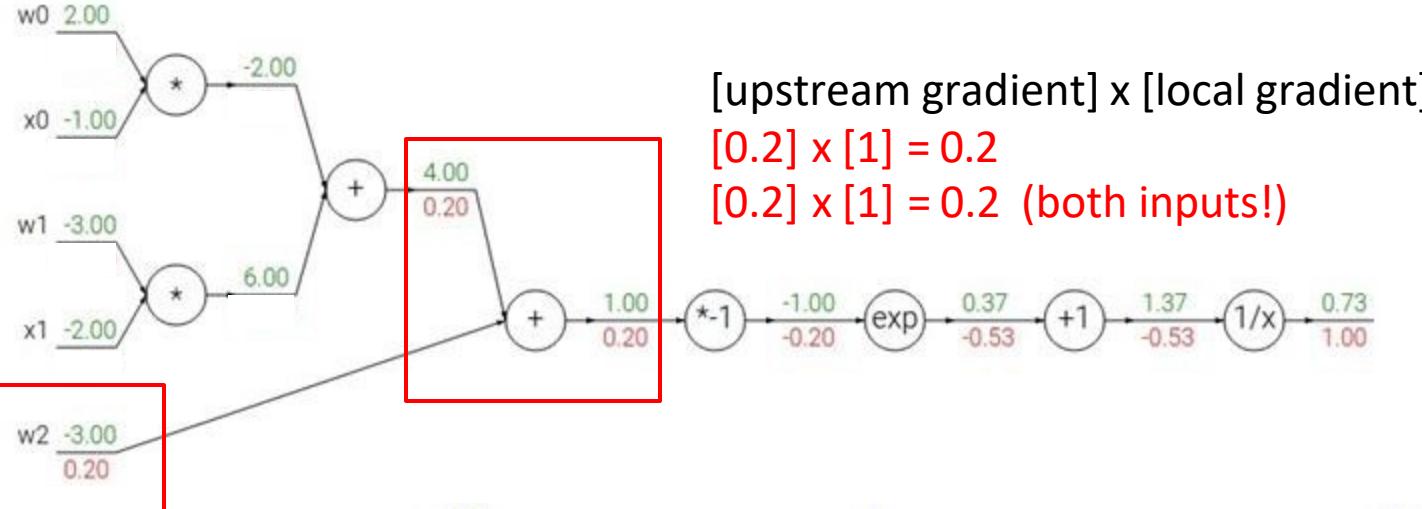
$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

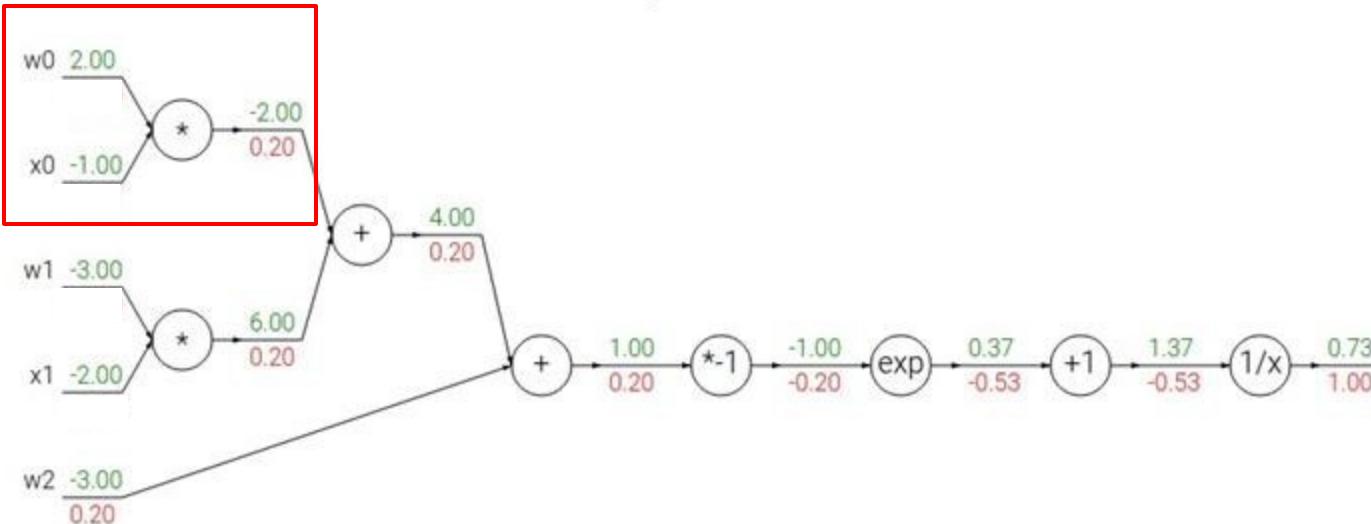
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

$\rightarrow$

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

$\rightarrow$

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

$\rightarrow$

$$\frac{df}{dx} = -1/x^2$$

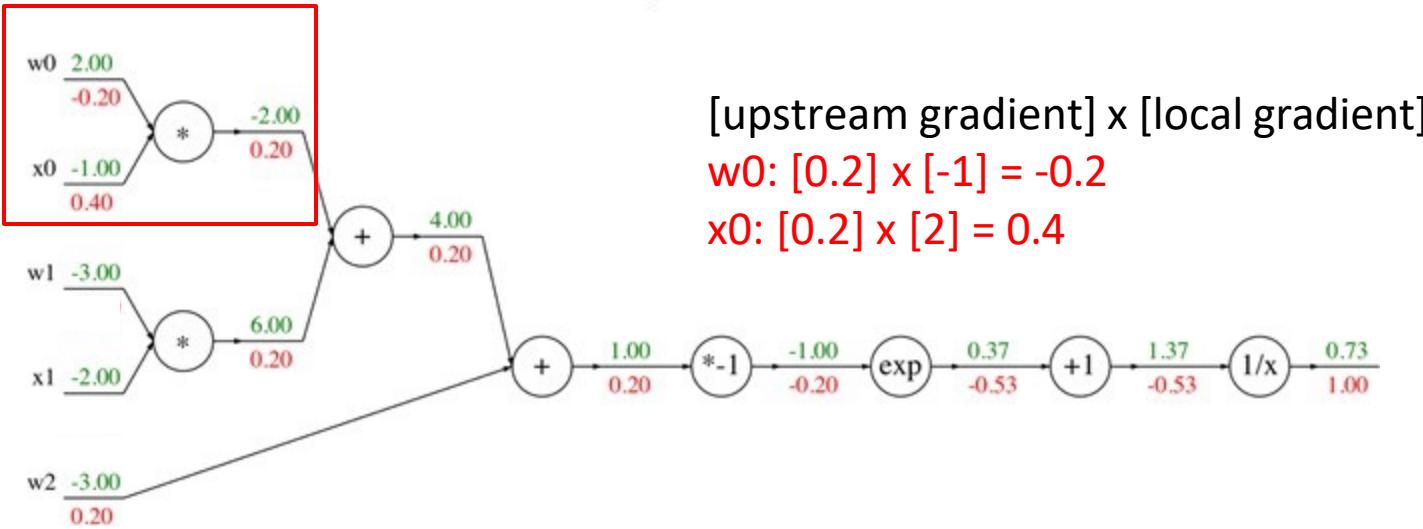
$$f_c(x) = c + x$$

$\rightarrow$

$$\frac{df}{dx} = 1$$

Another example:

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



$$f(x) = e^x$$

→

$$\frac{df}{dx} = e^x$$

$$f_a(x) = ax$$

→

$$\frac{df}{dx} = a$$

$$f(x) = \frac{1}{x}$$

→

$$\frac{df}{dx} = -1/x^2$$

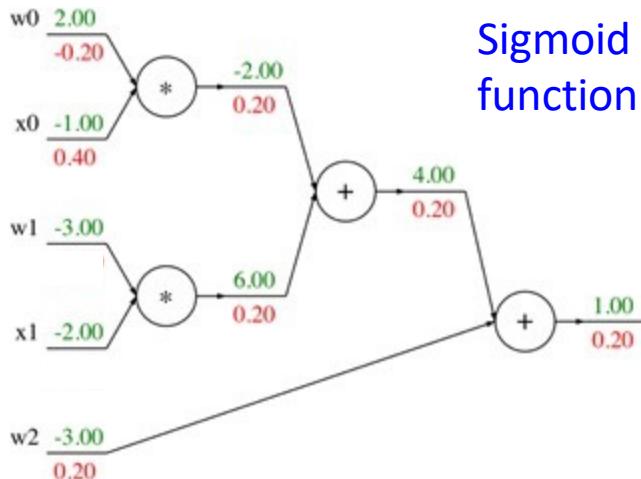
$$f_c(x) = c + x$$

→

$$\frac{df}{dx} = 1$$

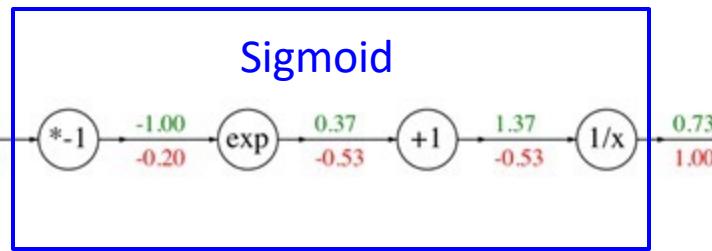
Another example:

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



Sigmoid  
function

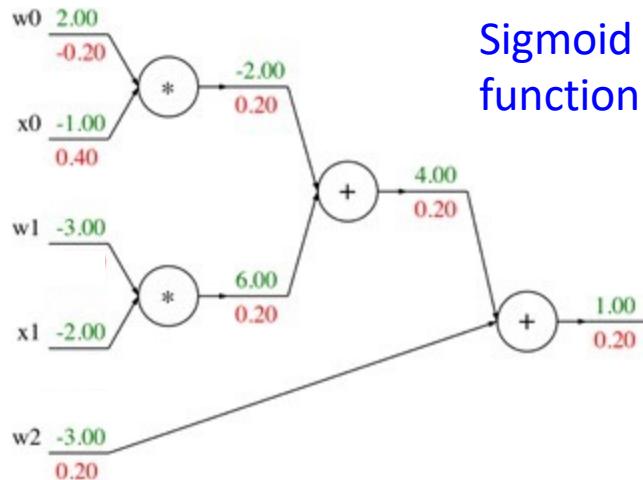
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$



Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

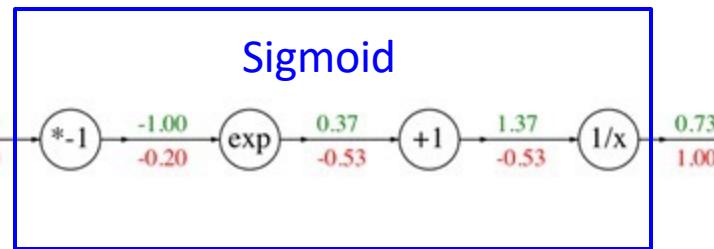
Another example:

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



Sigmoid  
function

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



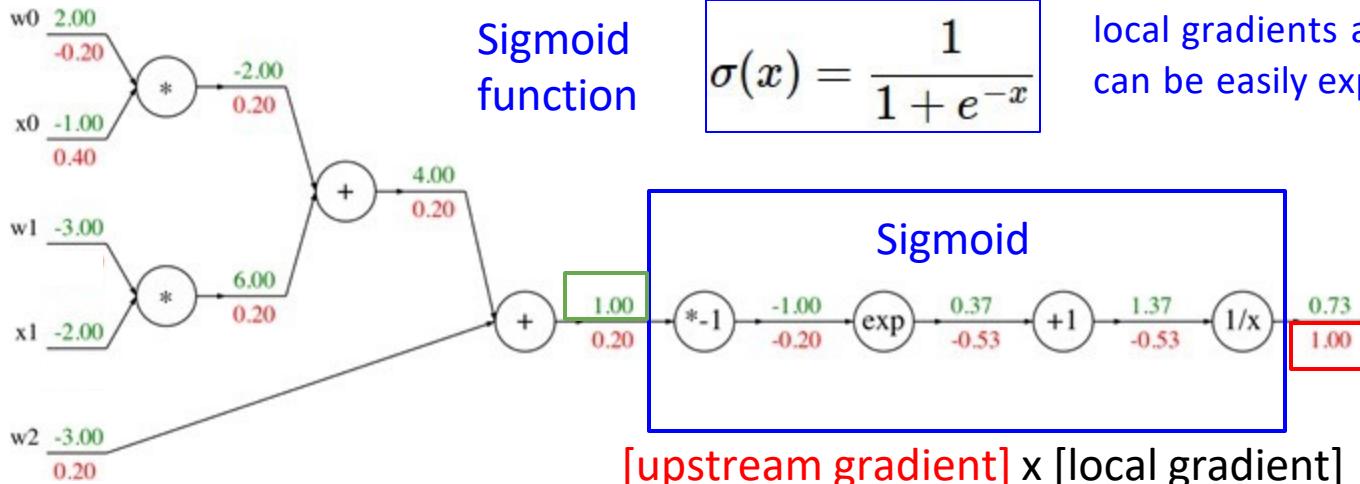
Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

Sigmoid local gradient:

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

Another example:

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



Sigmoid  
function

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

Sigmoid

[upstream gradient] x [local gradient]

$$[1.00] \times [(1 - 1/(1+e^{-1})) (1/(1+e^{-1}))] =$$

$$0.2$$

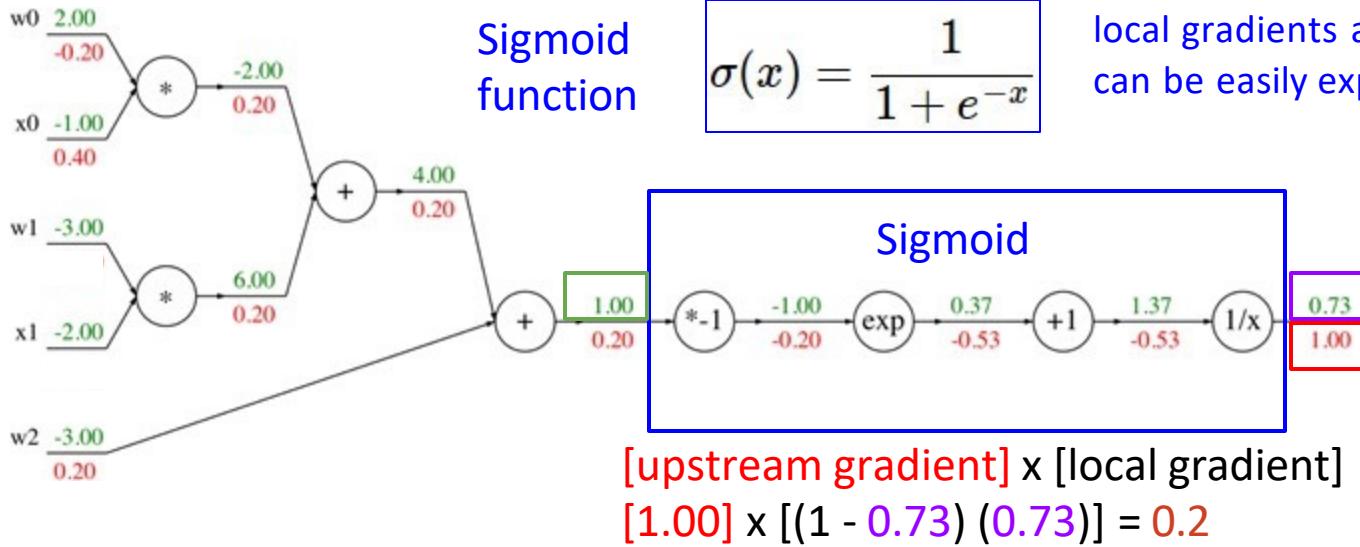
Sigmoid local  
gradient:

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

Another example:

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



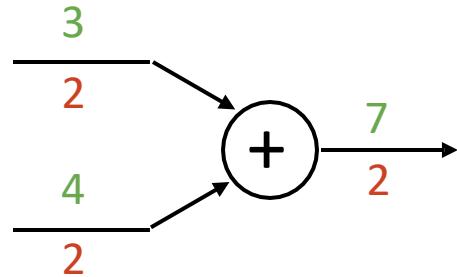
Sigmoid local  
gradient:

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

Computational graph representation may not be unique. Choose one where local gradients at each node can be easily expressed!

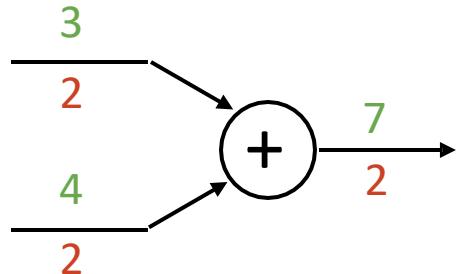
# Patterns in gradient flow

add gate: gradient distributor

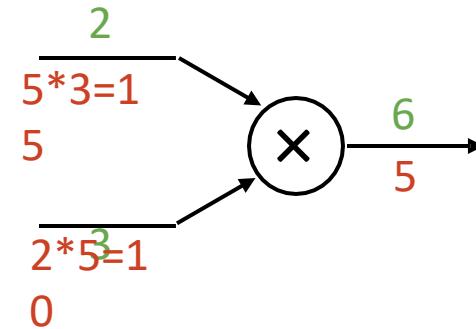


# Patterns in gradient flow

add gate: gradient distributor

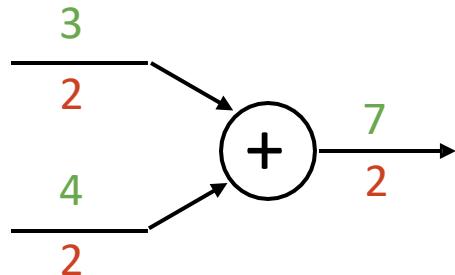


mul gate: “swap multiplier”

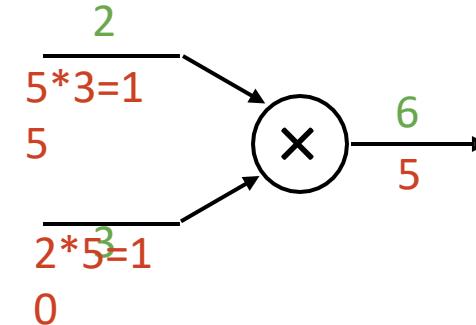


# Patterns in gradient flow

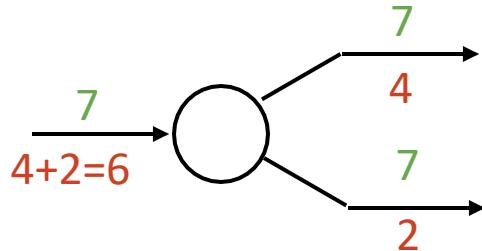
add gate: gradient distributor



mul gate: “swap multiplier”

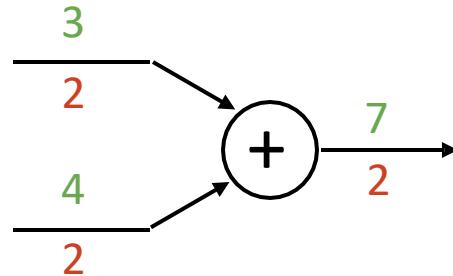


copy gate: gradient adder

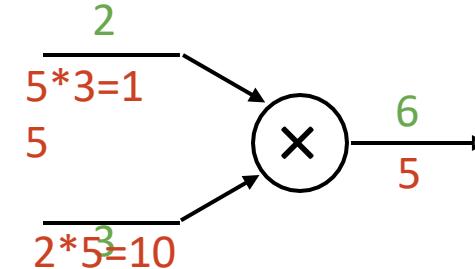


# Patterns in gradient flow

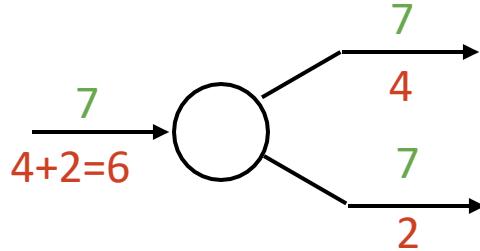
add gate: gradient distributor



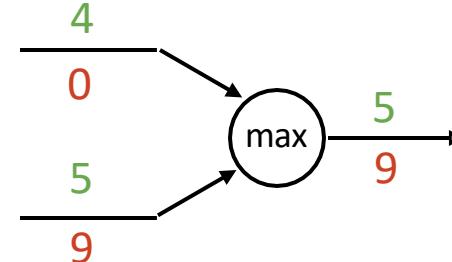
mul gate: “swap multiplier”



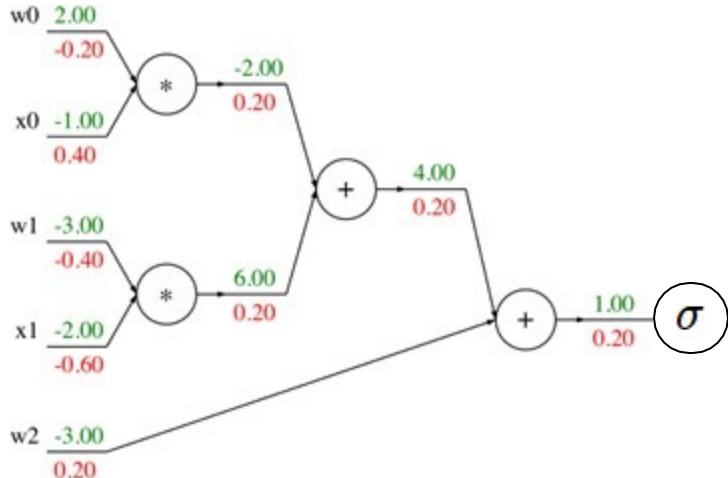
copy gate: gradient adder



max gate: gradient router



# Backprop Implementation: “Flat” code



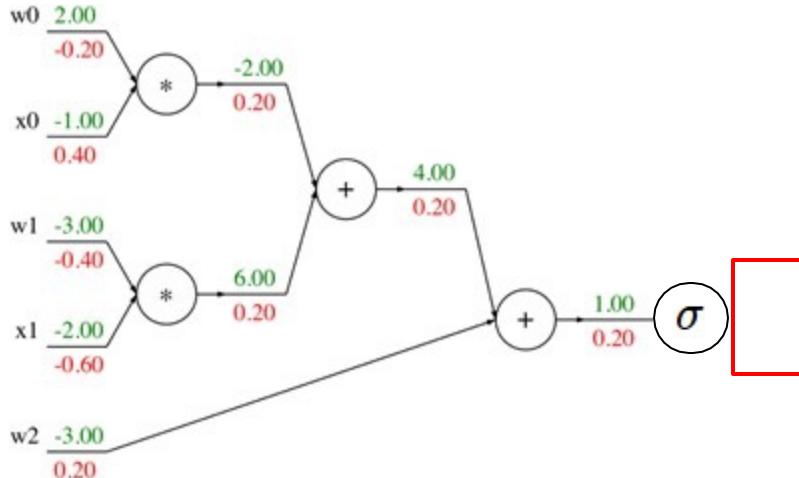
Forward pass:  
Compute output

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

Backward pass:  
Compute grads

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

# Backprop Implementation: “Flat” code



Forward pass:  
Compute output

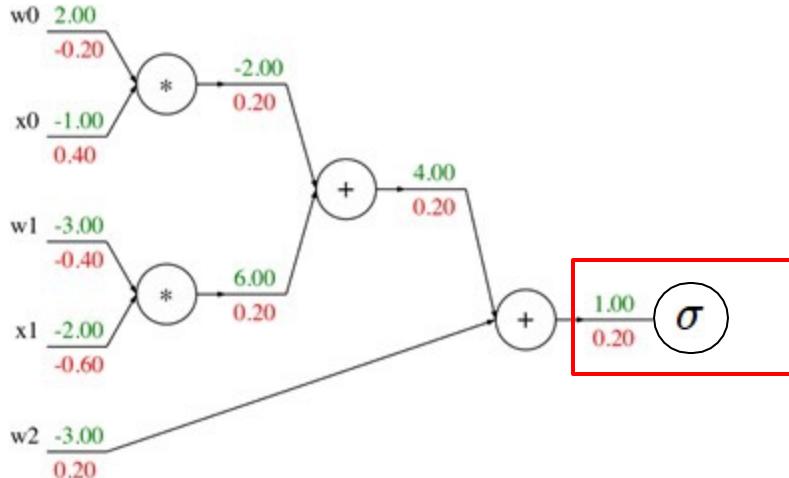
```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

Base case

grad\_L = 1.0

```
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

# Backprop Implementation: “Flat” code



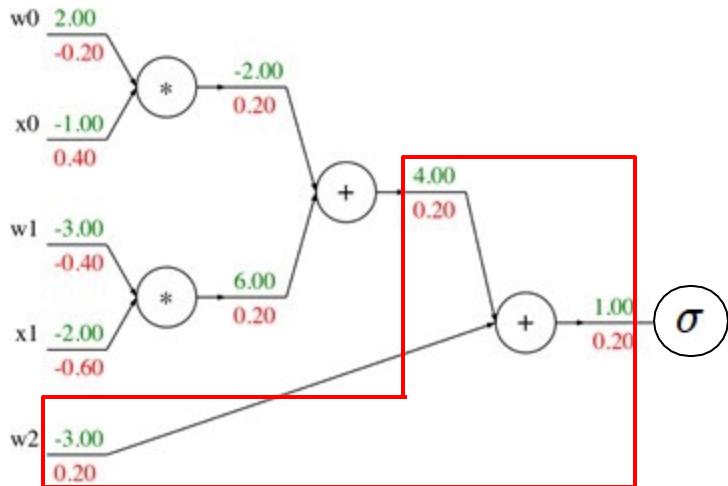
Forward pass:  
Compute output

Sigmoid

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

# Backprop Implementation: “Flat” code



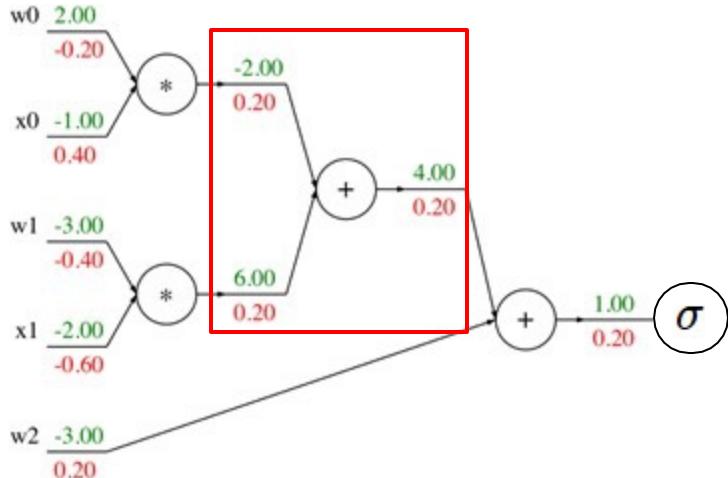
Forward pass:  
Compute output

Add gate

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

# Backprop Implementation: “Flat” code



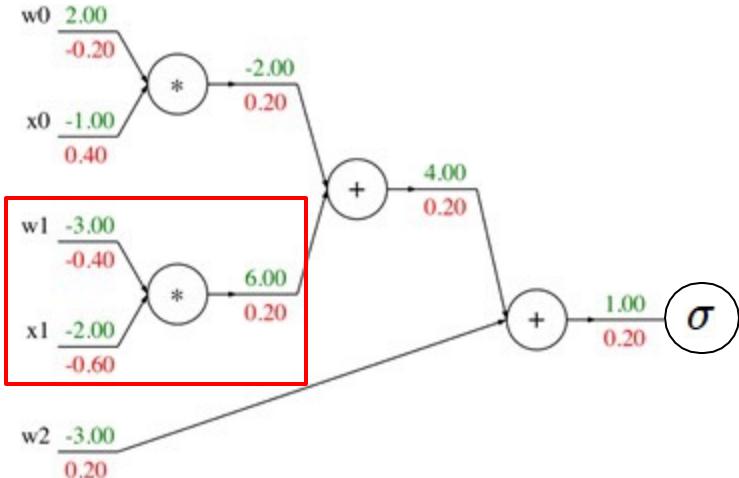
Forward pass:  
Compute output

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

Add gate

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

# Backprop Implementation: “Flat” code



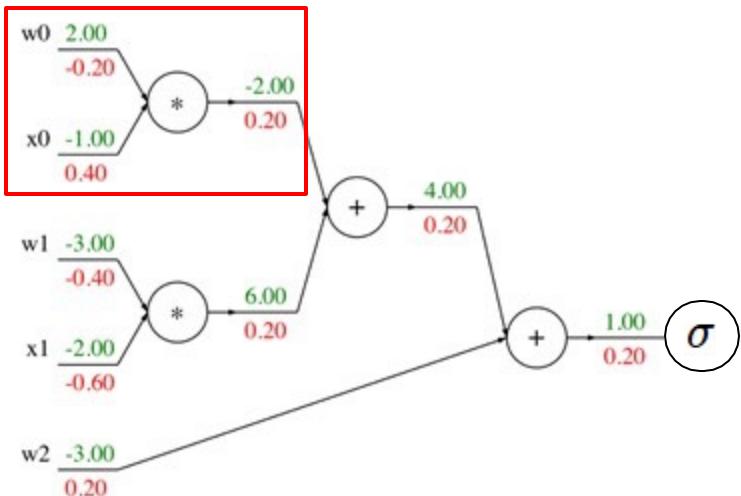
Forward pass:  
Compute output

```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

Multiply gate

# Backprop Implementation: “Flat” code



Forward pass:  
Compute output

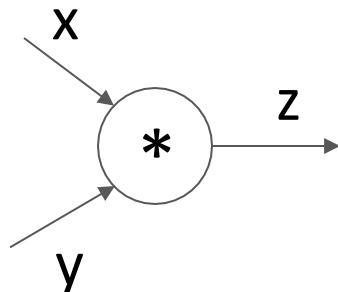
```
def f(w0, x0, w1, x1, w2):  
    s0 = w0 * x0  
    s1 = w1 * x1  
    s2 = s0 + s1  
    s3 = s2 + w2  
    L = sigmoid(s3)
```

```
grad_L = 1.0  
grad_s3 = grad_L * (1 - L) * L  
grad_w2 = grad_s3  
grad_s2 = grad_s3  
grad_s0 = grad_s2  
grad_s1 = grad_s2  
grad_w1 = grad_s1 * x1  
grad_x1 = grad_s1 * w1  
grad_w0 = grad_s0 * x0  
grad_x0 = grad_s0 * w0
```

Multiply gate

# Modularized implementation: forward / backward API

Gate / Node / Function object: Actual PyTorch code



( $x, y, z$  are scalars)

```
class Multiply(torch.autograd.Function):  
    @staticmethod  
    def forward(ctx, x, y):  
        ctx.save_for_backward(x, y) ←  
        z = x * y  
        return z  
    @staticmethod  
    def backward(ctx, grad_z): ←  
        x, y = ctx.saved_tensors  
        grad_x = y * grad_z # dz/dx * dL/dz  
        grad_y = x * grad_z # dz/dy * dL/dz  
        return grad_x, grad_y
```

Need to cache some values for use in backward

Upstream gradient

Multiply upstream and local gradients

# Example: PyTorch operators

pytorch / pytorch		
Code	Issues	Projects
<a href="#">Code</a>	<a href="#">Issues 2,288</a>	<a href="#">Pull requests 587</a>
<a href="#">Projects 4</a>	<a href="#">Wiki</a>	<a href="#">Insights</a>
<a href="#">Tree: 853938861</a> • <a href="#">pytorch / aten / src / THNN / generic /</a>	<a href="#">Close new file</a> <a href="#">Upload file</a> <a href="#">Find file</a> <a href="#">History</a>	
<a href="#">xzyang and facebook-github-bot Canonicalize all includes in PyTorch. (#14849)</a>	Latest commit 5121c7d8 on Dec 8, 2018	
-		
<a href="#">AbsCriterion.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">BCECriterion.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">ClassNLLCriterion.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">CrossEntropyLoss.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">ELU.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">FeaturePPPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">GatedLinearUnit.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">HardTanh.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">L1Distance.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">IndexLinear.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">LeakyReLU.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">LogSigmoid.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">MSELoss.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">MultiLabelMarginCriterion.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">MultiMarginCriterion.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">ReLU2.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">Sigmoid.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SmoothL1Criterion.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SoftMarginCriterion.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SoftPlus.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SoftShrink.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SparseLinear.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialAdaptiveAveragePooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialAdaptiveMaxPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialAveragePooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialConvolutionReLU.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialConvolutionRM.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialDilatedConvolution.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialGradientPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialFractionalMaxPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialFullConvolution.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialMaxPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialReflectionPadding.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialRepucesenPadding.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">SpatialUpSampling3DForward.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">THNN.h</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">Tanh.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">TemporalReflectionPadding.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">TemporalReplicationPadding.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">TemporalRowConvolution.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">TemporalUpSamplingInvert.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">TemporalUpSamplingNearest.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">TemporalUpSamplingNearest3D.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricAdaptiveAveragePooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricAdaptiveMaxPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricAveragePooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricConvolutionReLU.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricDilatedConvolution.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricDilatedMaxPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricFractionalMaxPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricFullConvolution.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricMaxPooling.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricReflectionPadding.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">VolumetricUpSampling3DForward.c</a>	Canonicalize all includes in PyTorch. (#14849)	4 months ago
<a href="#">linear_upsampling.h</a>	Implement nn.functional.interpolate based on upsample. (#14850)	9 months ago
<a href="#">pooling_shape.h</a>	Use integer math to compute output size of pooling operations. (#14850)	4 months ago
<a href="#">united.c</a>	Canonicalize all includes in PyTorch. (#14850)	4 months ago

# PyTorch sigmoid layer

```
1 #ifndef TH_GENERIC_FILE
2 #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3 #else
4
5 void THNN_(Sigmoid_updateOutput)(
6     THNNState *state,
7     THTensor *input,
8     THTensor *output)
9 {
10    THTensor_(sigmoid)(output, input);
11 }
12
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
26
27 #endif
```

Forward

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

[Source](#)

# PyTorch sigmoid layer

```
1 #ifndef TH_GENERIC_FILE
2 #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3 #else
4
5 void THNN_(Sigmoid_updateOutput)(
6     THNNState *state,
7     THTensor *input,
8     THTensor *output)
9 {
10    THTensor_(sigmoid)(output, input);
11 }
12
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
26
27 #endif
```

Forward

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

```
static void sigmoid_kernel(TensorIterator& iter) {
    AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
        unary_kernel_vec(
            iter,
            [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp((-a)))); },
            [=](Vec256<scalar_t> a) {
                a = Vec256<scalar_t>((scalar_t)(0)) - a;
                a = a.exp();
                a = Vec256<scalar_t>((scalar_t)(1)) + a;
                a = a.reciprocal();
                return a;
            });
    });
}
```

Forward actually defined [elsewhere...](#)

**return (1 / (1 + std::exp((-a))));**

[Source](#)

# PyTorch sigmoid layer

```
1 #ifndef TH_GENERIC_FILE
2 #define TH_GENERIC_FILE "THNN/generic/Sigmoid.c"
3 #else
4
5 void THNN_(Sigmoid_updateOutput)(
6     THNNState *state,
7     THTensor *input,
8     THTensor *output)
9 {
10     THTensor_(sigmoid)(output, input);
11 }
```

Forward

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

```
13 void THNN_(Sigmoid_updateGradInput)(
14     THNNState *state,
15     THTensor *gradOutput,
16     THTensor *gradInput,
17     THTensor *output)
18 {
19     THNN_CHECK_NELEMENT(output, gradOutput);
20     THTensor_(resizeAs)(gradInput, output);
21     TH_TENSOR_APPLY3(scalar_t, gradInput, scalar_t, gradOutput, scalar_t, output,
22         scalar_t z = *output_data;
23         *gradInput_data = *gradOutput_data * (1. - z) * z;
24     );
25 }
26
27 #endif
```

```
static void sigmoid_kernel(TensorIterator& iter) {
    AT_DISPATCH_FLOATING_TYPES(iter.dtype(), "sigmoid_cpu", [&]() {
        unary_kernel_vec(
            iter,
            [=](scalar_t a) -> scalar_t { return (1 / (1 + std::exp(-a))); },
            [=](Vec256<scalar_t> a) {
                a = Vec256<scalar_t>((scalar_t)(0)) - a;
                a = a.exp();
                a = Vec256<scalar_t>((scalar_t)(1)) + a;
                a = a.reciprocal();
                return a;
            });
    });
}
```

Forward actually defined [elsewhere...](#)

Backward

$$(1 - \sigma(x)) \sigma(x)$$

[Source](#)

So far: backprop with scalars

What about vector-valued functions?

# Recap: Vector derivatives

Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If  $x$  changes by a small amount, how much will  $y$  change?

# Recap: Vector derivatives

Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If  $x$  changes by a small amount, how much will  $y$  change?

Vector to Scalar

$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is Gradient:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left( \frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n}$$

For each element of  $x$ , if it changes by a small amount then how much will  $y$  change?

# Recap: Vector derivatives

## Scalar to Scalar

$$x \in \mathbb{R}, y \in \mathbb{R}$$

Regular derivative:

$$\frac{\partial y}{\partial x} \in \mathbb{R}$$

If  $x$  changes by a small amount, how much will  $y$  change?

## Vector to Scalar

$$x \in \mathbb{R}^N, y \in \mathbb{R}$$

Derivative is Gradient:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^N \quad \left( \frac{\partial y}{\partial x} \right)_n = \frac{\partial y}{\partial x_n}$$

For each element of  $x$ , if it changes by a small amount then how much will  $y$  change?

## Vector to Vector

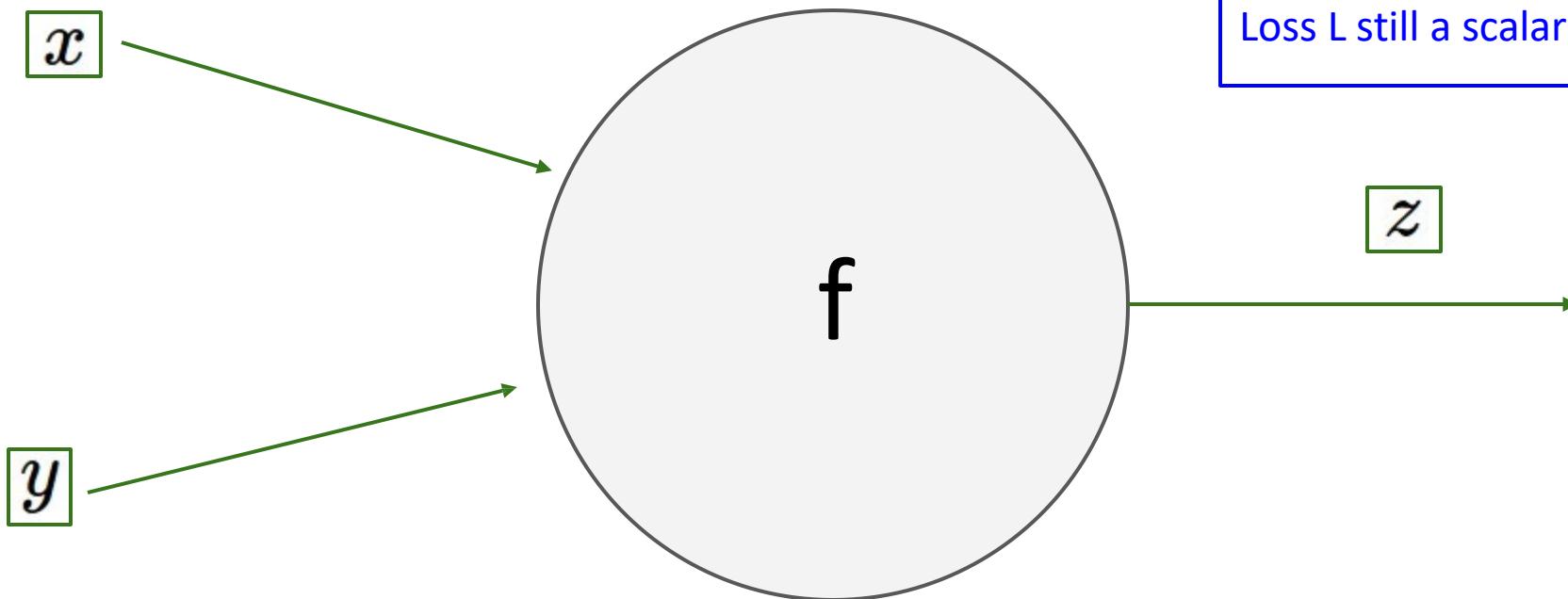
$$x \in \mathbb{R}^N, y \in \mathbb{R}^M$$

Derivative is Jacobian:

$$\frac{\partial y}{\partial x} \in \mathbb{R}^{N \times M} \quad \left( \frac{\partial y}{\partial x} \right)_{n,m} = \frac{\partial y_m}{\partial x_n}$$

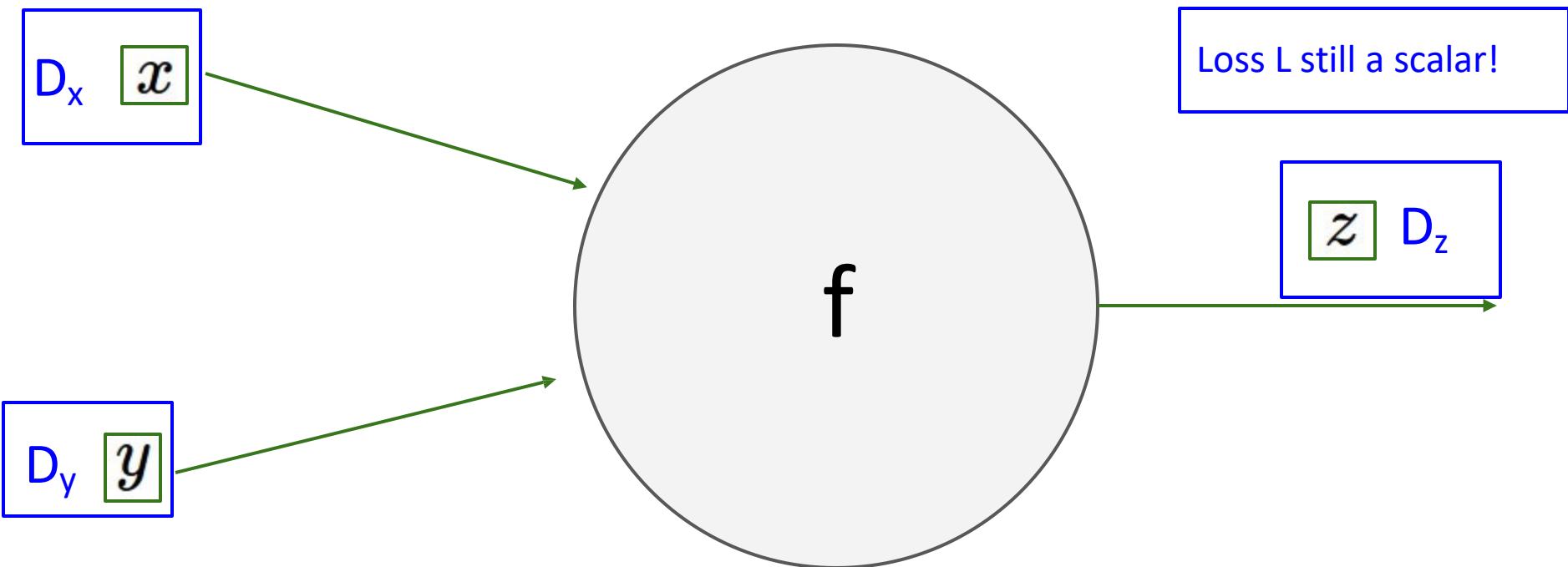
For each element of  $x$ , if it changes by a small amount then how much will each element of  $y$  change?

# Backprop with Vectors

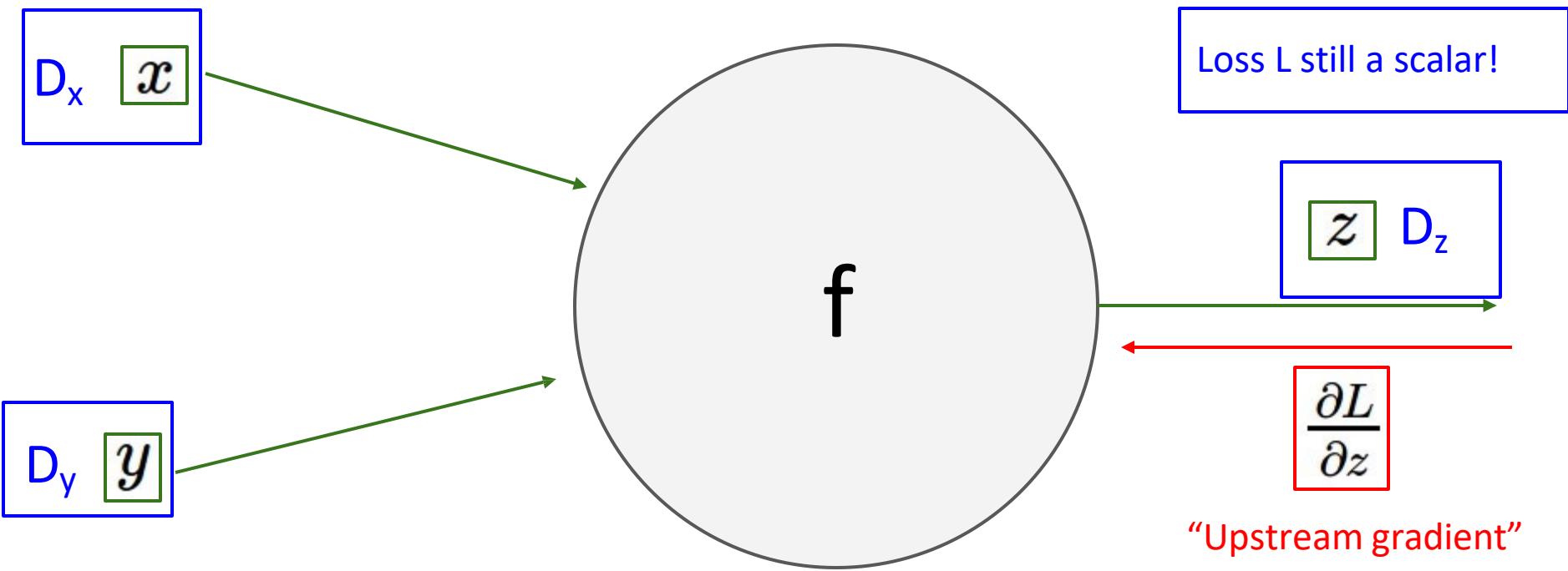


Loss L still a scalar!

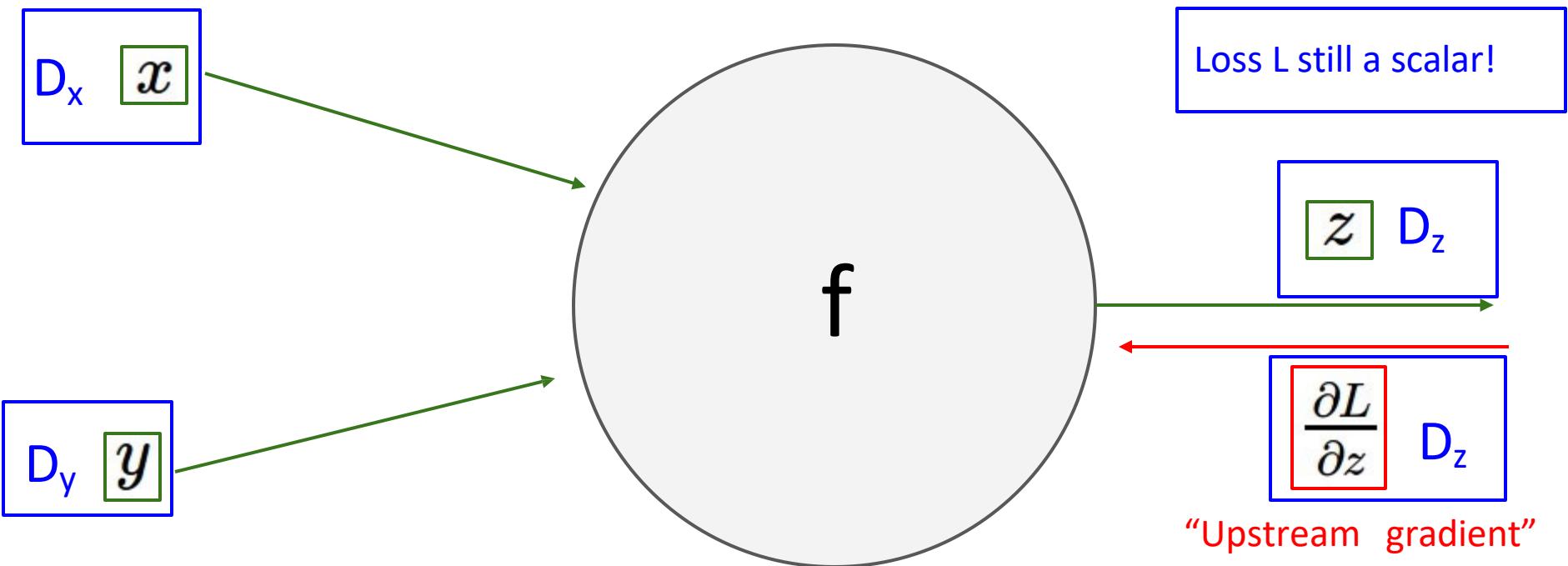
# Backprop with Vectors



# Backprop with Vectors

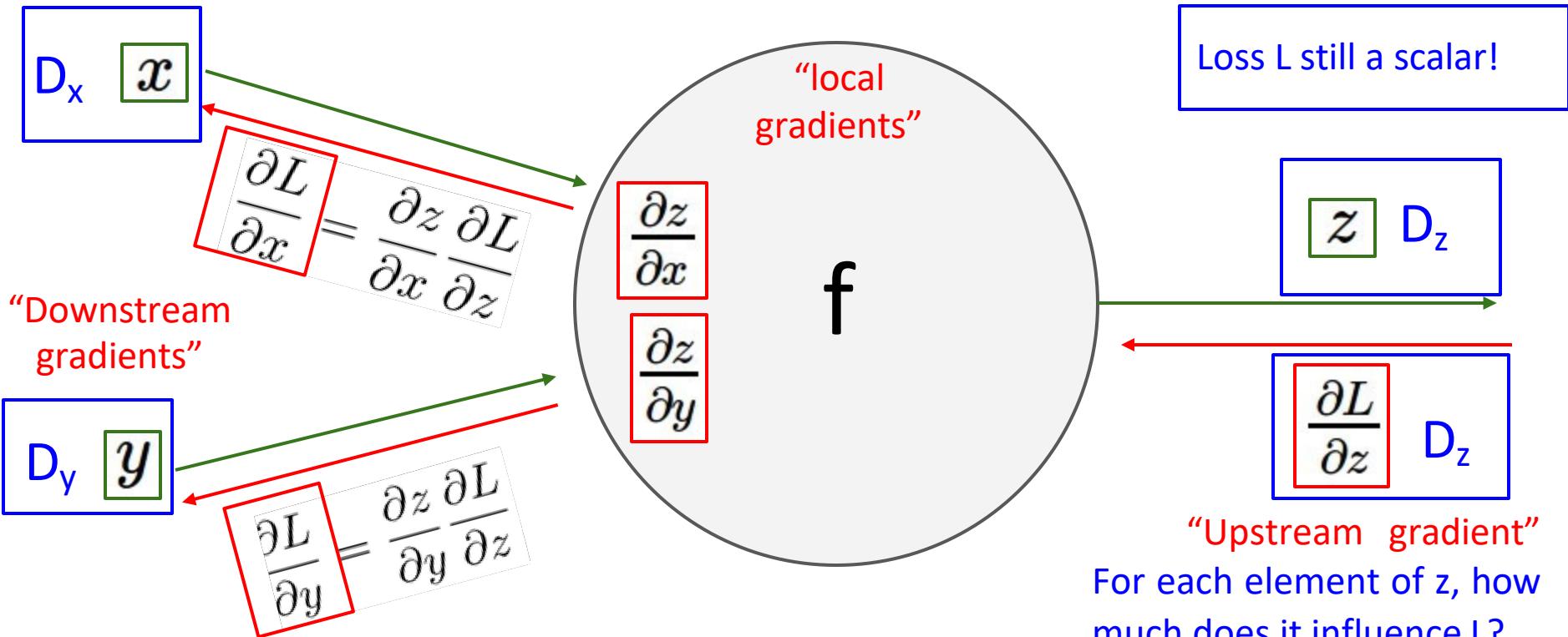


# Backprop with Vectors

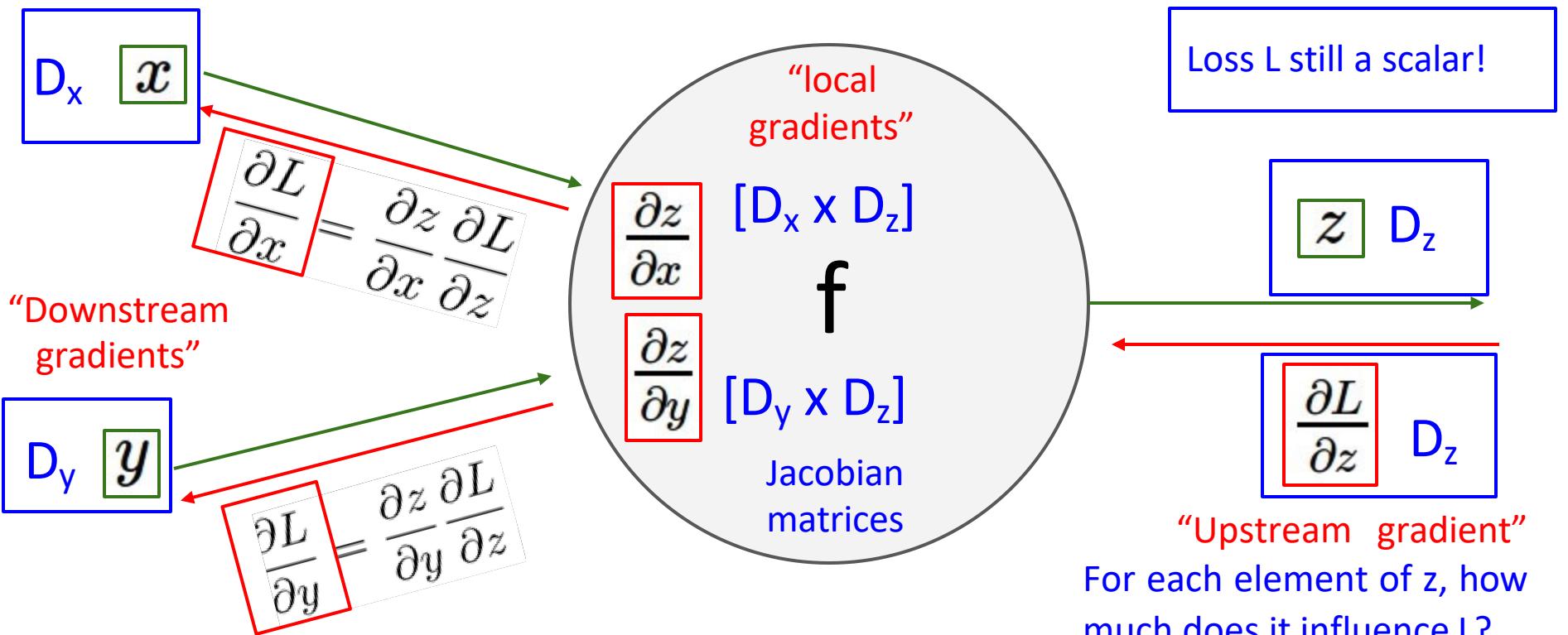


“Upstream gradient”  
For each element of  $z$ , how  
much does it influence  $L$ ?

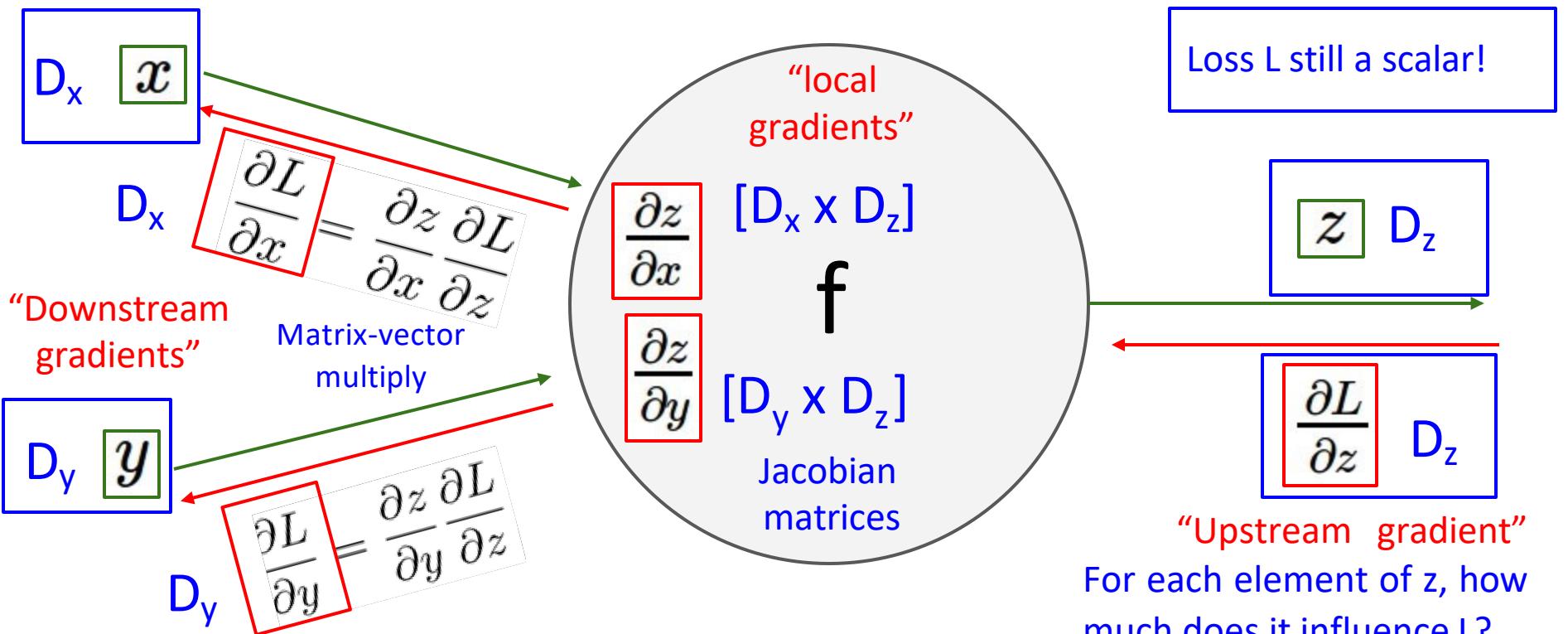
# Backprop with Vectors



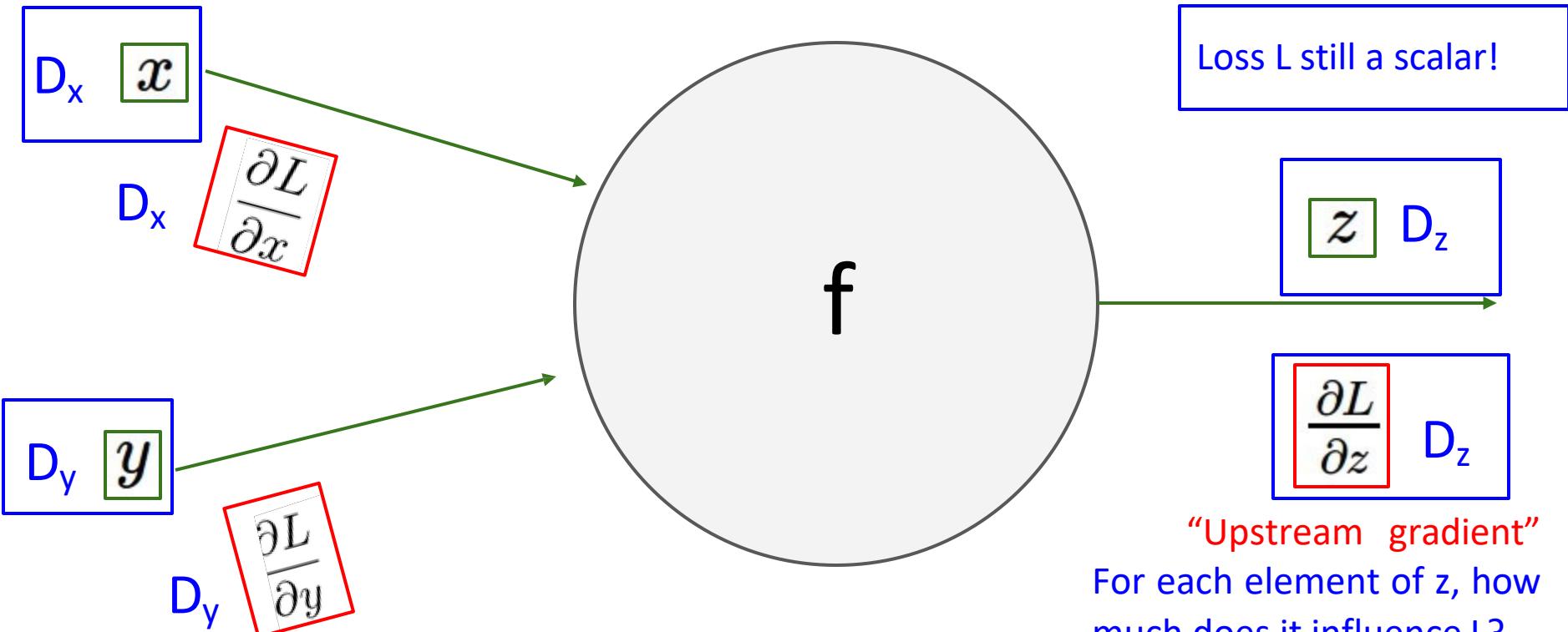
# Backprop with Vectors



# Backprop with Vectors



Gradients of variables wrt loss have same dims as the original variable



“Upstream gradient”  
For each element of  $z$ , how  
much does it influence  $L$ ?

# Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow$$

4D output z:

$$\longrightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

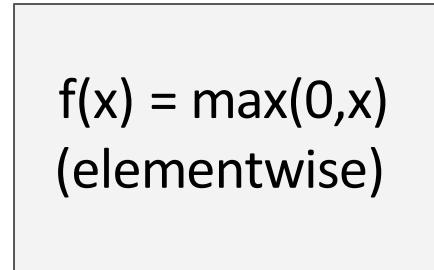
$$f(x) = \max(0, x)$$

(elementwise)

# Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow$$



4D output z:

$$\longrightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D  $dL/dz$ :

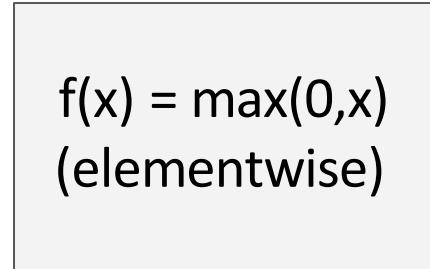
$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \longleftarrow$$

Upstream  
gradient

# Backprop with Vectors

4D input  $x$ :

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow$$



4D output  $z$ :

$$\longrightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

Jacobian  $\frac{\partial z}{\partial x}$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

4D  $\frac{\partial L}{\partial z}$ :

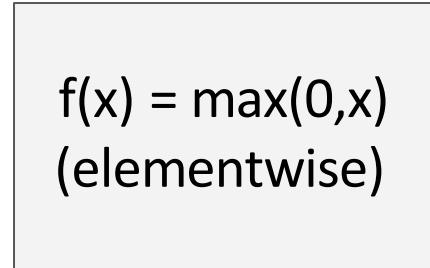
$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \longleftarrow$$

Upstream  
gradient

# Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow$$



4D output z:

$$\longrightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

[dz/dx] [dL/dz]

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

4D dL/dz:

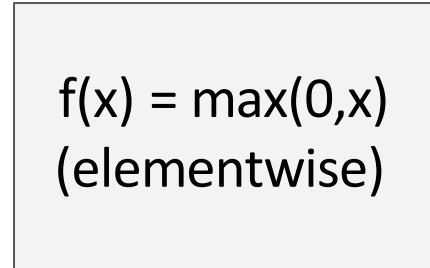
$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \leftarrow \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

Upstream  
gradient

# Backprop with Vectors

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow$$



4D output z:

$$\longrightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D  $dL/dx$ :

$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \longleftarrow$$

$[dz/dx] [dL/dz]$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

4D  $dL/dz$ :

$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \longleftarrow \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

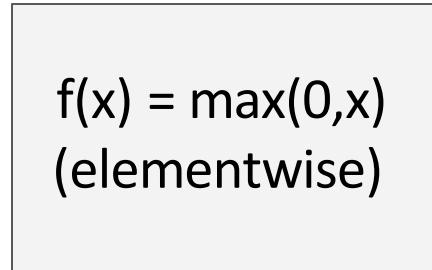
Upstream  
gradient

# Backprop with Vectors

Jacobian is sparse:  
off-diagonal entries  
always zero! Never  
explicitly form  
Jacobian -- instead  
use implicit  
multiplication

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow$$



4D output z:

$$\longrightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D  $dL/dx$ :

$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \longleftarrow$$

$[dz/dx] [dL/dz]$

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

4D  $dL/dz$ :

$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \longleftarrow \begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix}$$

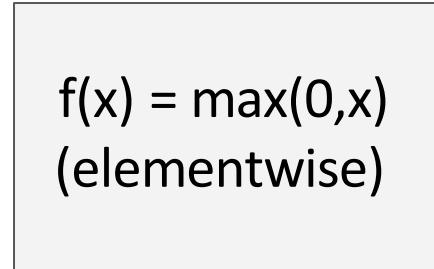
Upstream  
gradient

# Backprop with Vectors

Jacobian is sparse:  
off-diagonal entries  
always zero! Never  
explicitly form  
Jacobian -- instead  
use implicit  
multiplication

4D input x:

$$\begin{bmatrix} 1 \\ -2 \\ 3 \\ -1 \end{bmatrix} \longrightarrow$$



4D output z:

$$\longrightarrow \begin{bmatrix} 1 \\ 0 \\ 3 \\ 0 \end{bmatrix}$$

4D  $dL/dx$ :

$$\begin{bmatrix} 4 \\ 0 \\ 5 \\ 0 \end{bmatrix} \leftarrow$$

$$\left( \frac{\partial L}{\partial x} \right)_i = \begin{cases} \left( \frac{\partial L}{\partial z} \right)_i & \text{if } x_i > 0 \\ 0 & \text{otherwise} \end{cases}$$

$[dz/dx]$   $[dL/dz]$

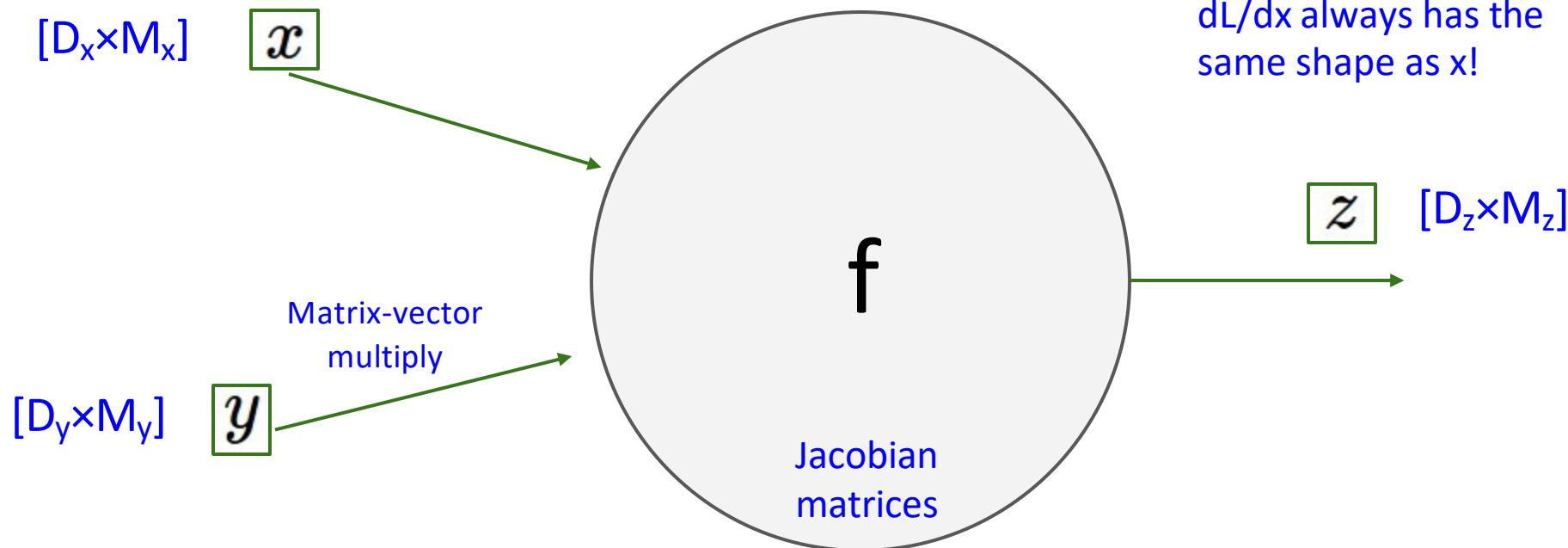
4D  $dL/dz$ :

$$\begin{bmatrix} 4 \\ -1 \\ 5 \\ 9 \end{bmatrix} \leftarrow$$

Upstream  
gradient

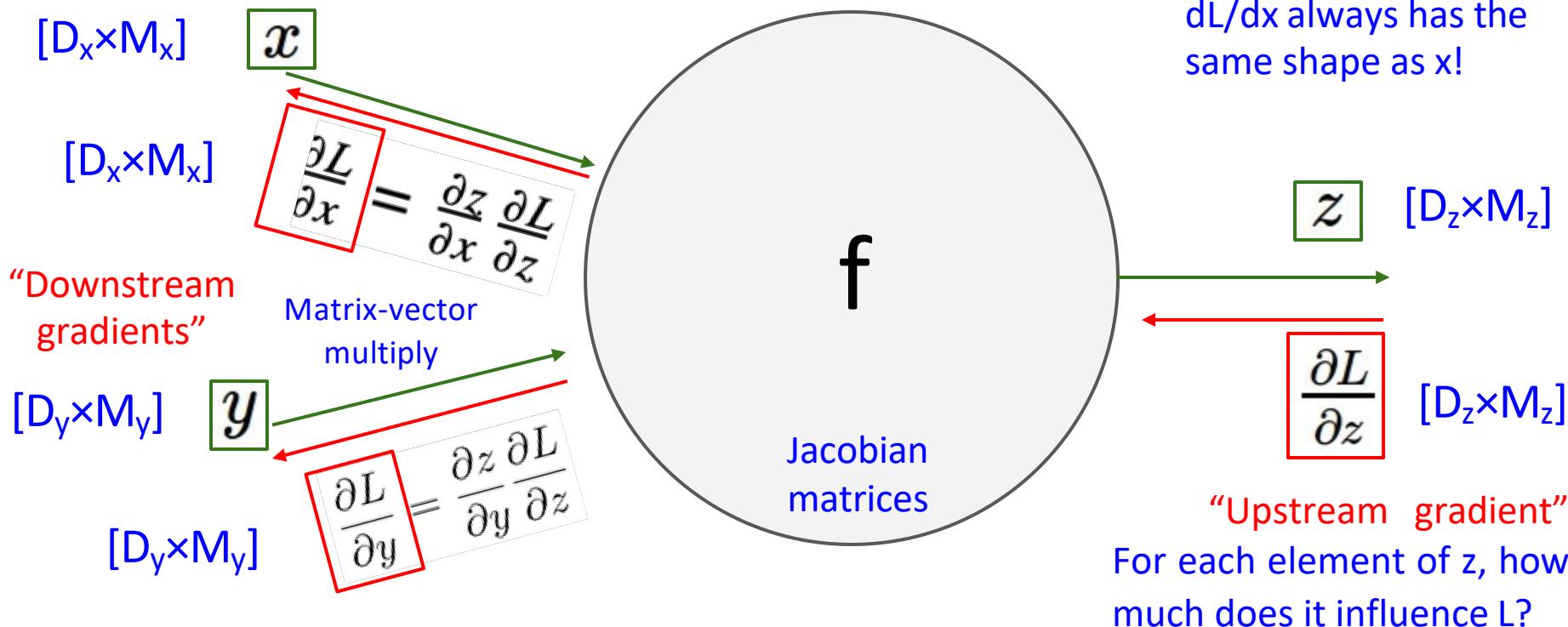
# Backprop with Matrices (or Tensors)

Loss L still a scalar!



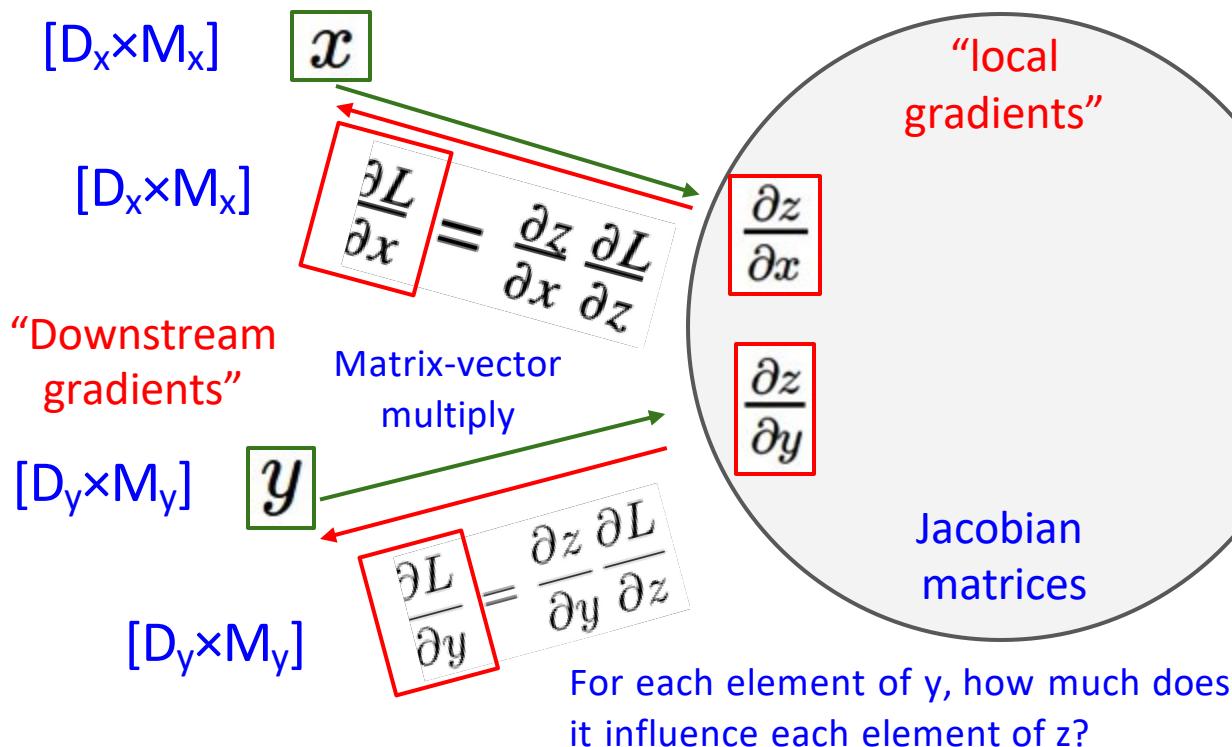
# Backprop with Matrices (or Tensors)

Loss L still a scalar!



# Backprop with Matrices (or Tensors)

Loss L still a scalar!

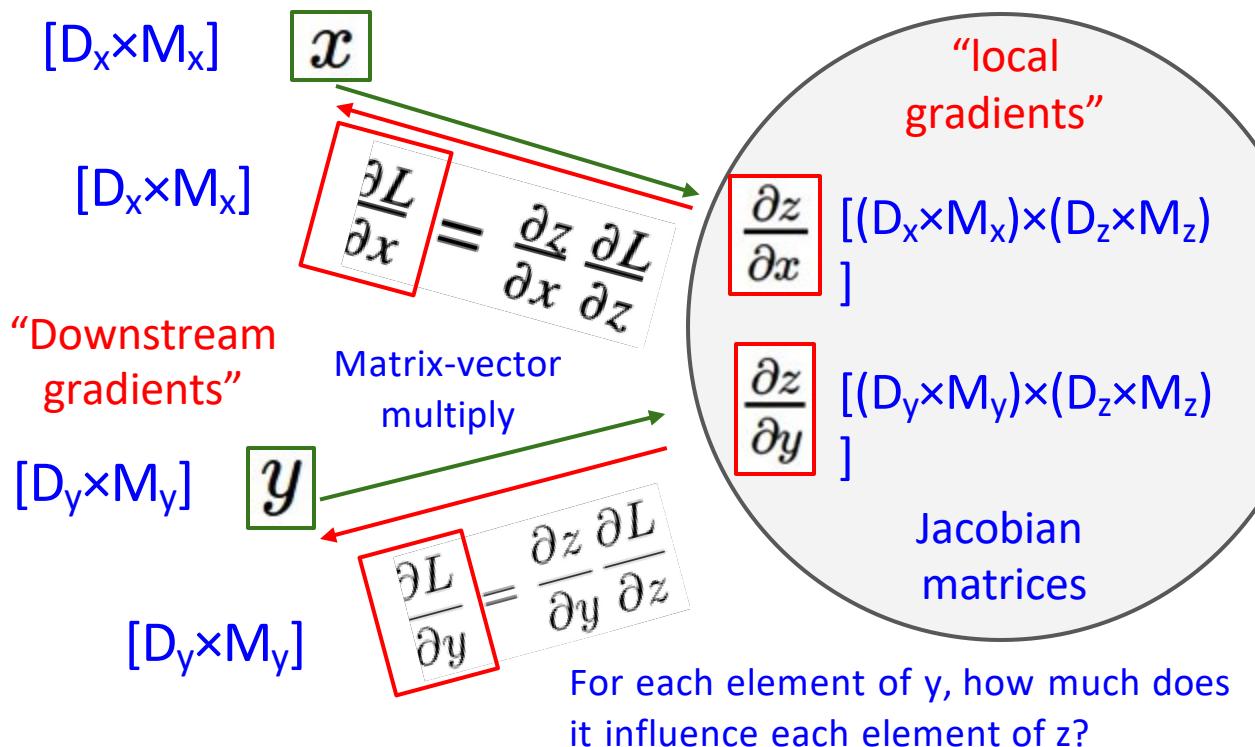


dL/dx always has the same shape as x!

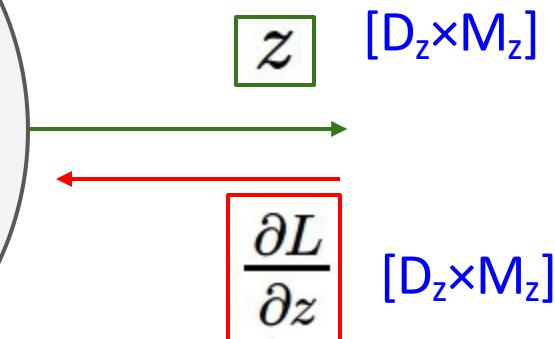
"Upstream gradient"  
For each element of  $z$ , how much does it influence L?

# Backprop with Matrices (or Tensors)

Loss L still a scalar!



$dL/dx$  always has the same shape as  $x$ !



"Upstream gradient"  
For each element of  $z$ , how much does it influence  $L$ ?

# Backprop with Matrices

x: [N×D]

[ 2 1 -3 ]

[ -3 4 2 ]

w: [D×M]

[ 3 2 1 -  
1 ]

[ 2 1 3 ]

2 ]

[ 3 2 1 -  
2 ]



Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$



y: [N×M]

[13 9 -2 -6 ]

[ 5 2 17 1 ]



dL/dy: [N×M]

[ 2 3 -3 9 ]

[ -8 1 4 6 ]



Also see derivation in the course notes:

<http://cs231n.stanford.edu/handouts/linear-backprop.pdf>

# Backprop with Matrices

$x: [N \times D]$

$$\begin{bmatrix} 2 & 1 & -3 \end{bmatrix}$$

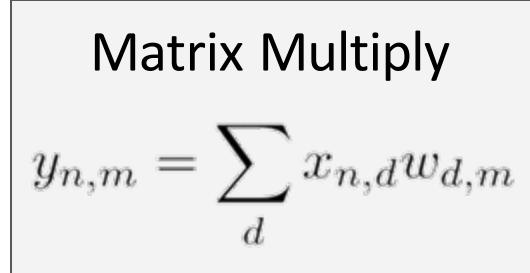
$$\begin{bmatrix} -3 & 4 & 2 \end{bmatrix}$$

$w: [D \times M]$

$$\begin{bmatrix} 3 & 2 & 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 2 & 1 & 3 \\ 2 \end{bmatrix}$$

$$\begin{bmatrix} 3 & 2 & 1 \\ 2 \end{bmatrix}$$



$y: [N \times M]$

$$\begin{bmatrix} 13 & 9 & -2 & -6 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 2 & 17 & 1 \end{bmatrix}$$



$dL/dy: [N \times M]$

$$\begin{bmatrix} 2 & 3 & -3 & 9 \end{bmatrix}$$

$$\begin{bmatrix} -8 & 1 & 4 & 6 \end{bmatrix}$$

Jacobians:

$dy/dx: [(N \times D) \times (N \times M)]$

$dy/dw: [(D \times M) \times (N \times M)]$

For a neural net we may have

$N=64, D=M=4096$

Each Jacobian takes  $\sim 256$  GB of memory!

Must work with them implicitly!

# Backprop with Matrices

$x: [N \times D]$

$$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$w: [D \times M]$

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & 3 & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

Q: What parts of  $y$  are affected by one element of  $x$ ?

$y: [N \times M]$

$$\begin{bmatrix} 13 & 9 & -2 & -6 \end{bmatrix}$$

$$\begin{bmatrix} 5 & 2 & 17 & 1 \end{bmatrix}$$

$dL/dy: [N \times M]$

$$\begin{bmatrix} 2 & 3 & -3 & 9 \end{bmatrix}$$

$$\begin{bmatrix} -8 & 1 & 4 & 6 \end{bmatrix}$$

# Backprop with Matrices

x: [N×D]

$$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & 3 & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

Q: What parts of y are affected by one element of x?  
A:  $x_{n,d}$  affects the whole row  $y_{n,:}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}$$

y:

$$\begin{bmatrix} 1 & 3 & N & M & -6 \\ 1 & 5 & 2 & 17 & 1 \\ \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \\ \end{bmatrix}$$

# Backprop with Matrices

x: [N×D]

$$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

w: [D×M]

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 2 & 1 & 3 & 2 \\ 3 & 2 & 1 & -2 \end{bmatrix}$$

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

y:

$$\begin{bmatrix} 1 & 3 & N & M & -6 \\ 1 & 5 & 2 & 17 & 1 \end{bmatrix}$$

dL/dy: [N×M]

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

Q: What parts of y are affected by one element of x?

A:  $x_{n,d}$  affects the whole row  $y_{n,:}$

Q: How much does  $x_{n,d}$  affect  $y_{n,m}$ ?

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}}$$

# Backprop with Matrices

$x: [N \times D]$

$$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$w: [D \times M]$

$$\begin{bmatrix} 3 & 2 & 1 & - \\ 1 & \square \\ 3 & 2 & 3 & - \\ 2 \end{bmatrix}$$



Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$



y:

$$\begin{bmatrix} 1 & 3 & N & M & -6 \\ 1 & 5 & 2 & 17 & 1 \\ \end{bmatrix}$$

$$\begin{bmatrix] & & & & \\ & & & & \end{bmatrix}$$

$dL/dy: [N \times M]$

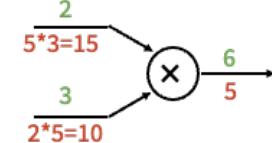
$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \end{bmatrix}$$

$$\begin{bmatrix] & & & & \\ & & & & \end{bmatrix}$$

Q: What parts of y are affected by one element of x?  
 A:  $x_{n,d}$  affects the whole row  $y_{n,:}$

Q: How much does  $x_{n,d}$  affect  $y_{n,m}$ ?  
 A:  $w_{d,m}$

mul gate: "swap multiplier"



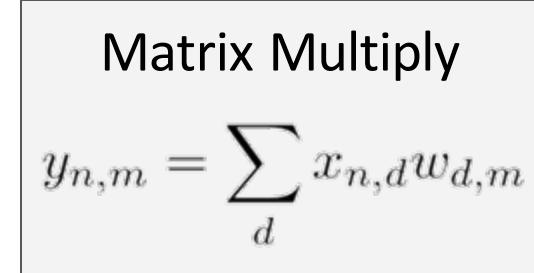
$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}$$

# Backprop with Matrices

x: [N×D]  
[ 2 1 -3 ]  
[ -3 4 2 ]

w: [D×M]  
[ 3 2 1 -  
1 ]          
[ 3 2 3 -2 ]  
[ 2 ]  
[N×D] [N×M]

$$\frac{\partial L}{\partial x} = \left( \frac{\partial L}{\partial y} \right) w^T$$



y:  
[ 1 3 N M -6 ]  
I 5 2 17 1  
]  
dL/dy: [N×M]  
[ 2 3 -3 9 ]  
[ -8 1 4 6 ]

Q: What parts of y are affected by one element of x?  
A:  $x_{n,d}$  affects the whole row  $y_{n,:}$ .

Q: How much does  $x_{n,d}$  affect  $y_{n,m}$ ?  
A:  $w_{d,m}$

$$\frac{\partial L}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} \frac{\partial y_{n,m}}{\partial x_{n,d}} = \sum_m \frac{\partial L}{\partial y_{n,m}} w_{d,m}$$

# Backprop with Matrices

$x: [N \times D]$

$$\begin{bmatrix} 2 & 1 & -3 \\ -3 & 4 & 2 \end{bmatrix}$$

$w: [D \times M]$

$$\begin{bmatrix} 3 & 2 & 1 & -1 \\ 1 & \square \\ 3 & 2 & 3 & -2 \end{bmatrix}$$

2

[N×D] [N×M]

[M×D]

$$\frac{\partial L}{\partial x} = \left( \frac{\partial L}{\partial y} \right) w^T$$

Matrix Multiply

$$y_{n,m} = \sum_d x_{n,d} w_{d,m}$$

By similar logic:

[D×M] [D×N]

$$\frac{\partial L}{\partial w} = x^T \left( \frac{\partial L}{\partial y} \right)$$

y:

$$\begin{bmatrix} 1 & 3 & N & M & -6 \\ I & 5 & 2 & 17 & 1 \\ \end{bmatrix}$$

$dL/dy: [N \times M]$

$$\begin{bmatrix} 2 & 3 & -3 & 9 \\ -8 & 1 & 4 & 6 \\ \end{bmatrix}$$

These formulas are easy to remember: they are the only way to make shapes match up!

# Summary for today:

- (Fully-connected) Neural Networks are stacks of linear functions and nonlinear activation functions; they have much more representational power than linear classifiers
- backpropagation = recursive application of the chain rule along a computational graph to compute the gradients of all inputs/parameters/intermediates
- implementations maintain a graph structure, where the nodes implement the forward() / backward() API
- forward: compute result of an operation and save any intermediates needed for gradient computation in memory
- backward: apply the chain rule to compute the gradient of the loss function with respect to the inputs

# Next Time: Convolutional Neural Networks!

