Neural Networks and Deep Learning

By Ghazal Lalooha

Table of Contents

Introduction: image classification

image classification

We are given a discrete set of labels

Which category (label) does the input image belong to?

- Airplane
- Automobile
- Bird
- Cat
- deer
- Dog
- Frog
- Horse
- Ship
- Truck

Problem: Semantic distance

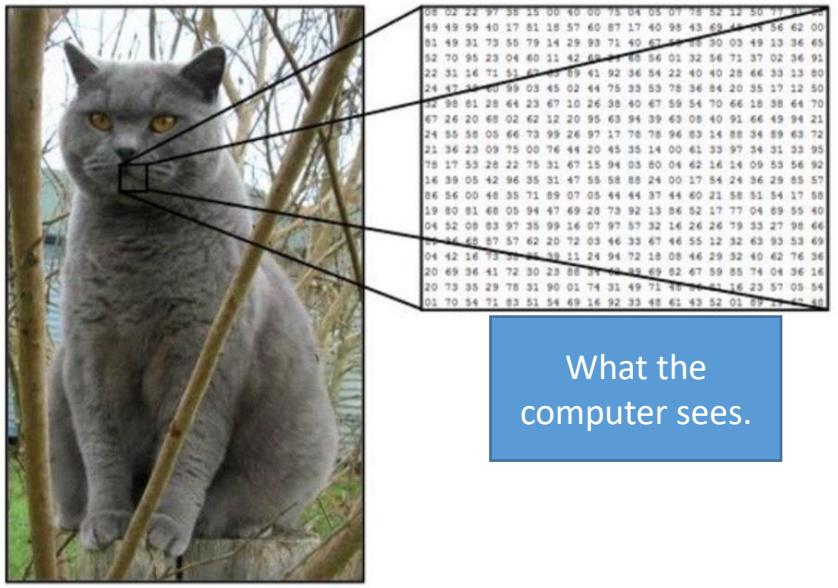
Image representation

- Images are stored in the computer as three-dimensional arrays of integers in the range [0, 255].

- Example:

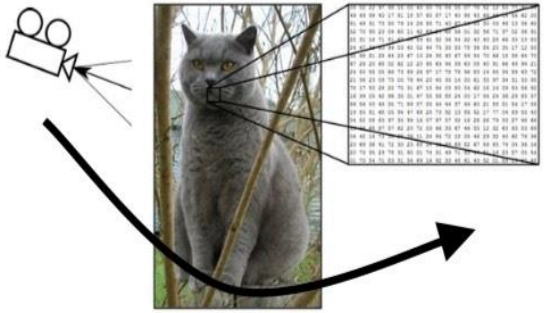
$$300 \times 100 \times 3$$

RGB



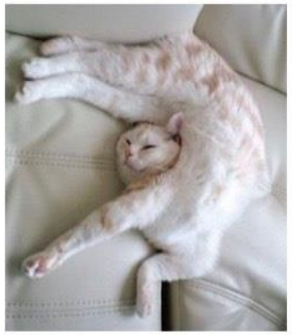
Challenges of image classification

Camera angle



lighting

Shape change



Obstruction

Blend with the background

Diversity in each class

An Image Classifier

- There is no obvious way to code a cat detection algorithm or other categories!
  - Unlike sorting a list of numbers

```
def predict(image):
    # ????
    return class_label
```

?

Airplane
Automobile
Bird
Cat
deer
Dog
Frog
Horse
Ship
Truck

Data-driven approach

- Data-driven approach:
  - Collect a collection of images and labels for each one.
  - Train a classifier to classify images using machine learning.
  - Evaluate the performance of the classifier on a set of test images.


```
def train(train_images, train_labels):
    # Build a model for images --> labels...
    return model

def predict(model, test_images):
    # predict test labels using the model...
    return test_labels
```

The first classifier

- Nearest Neighbor Classifier

memorize all of the training set's images and also the labels related to each of them.

Predict the label of the input test image according to the label of the most similar image in the training set.

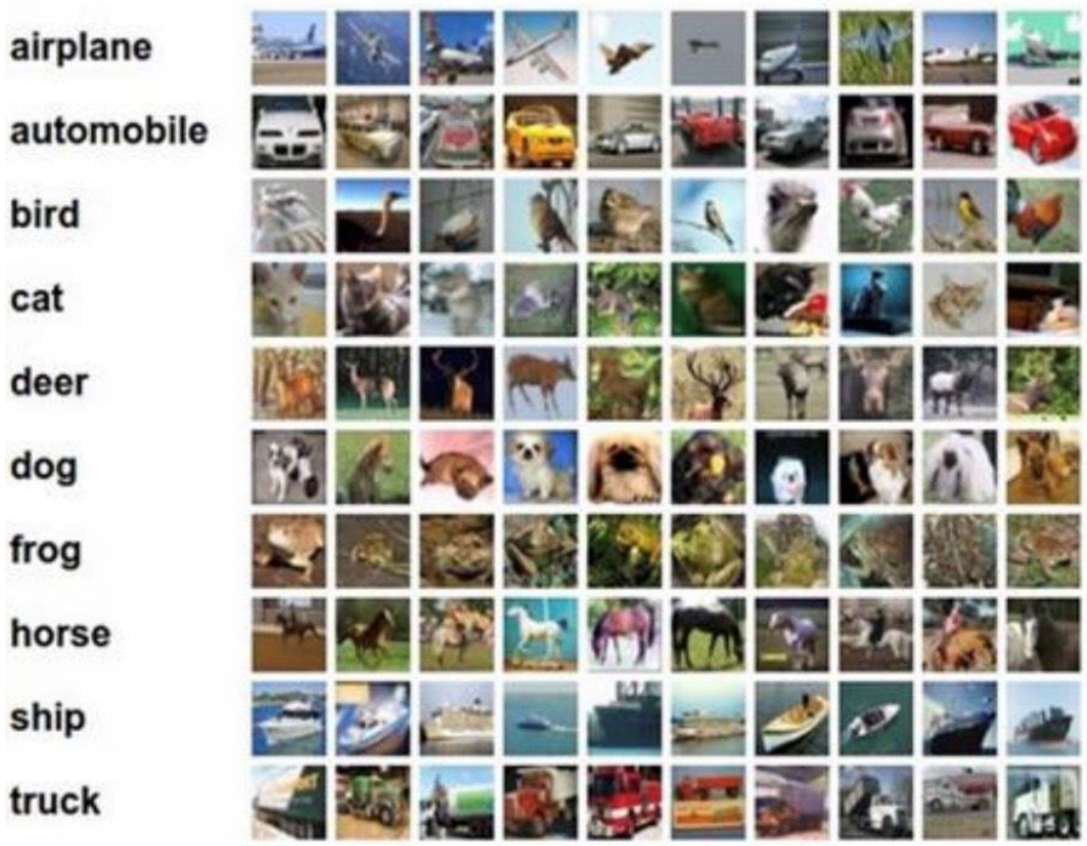
```
def train(train_images, train_labels):
    # Build a model for images --> labels...
    return model

def predict(model, test_images):
    # predict test labels using the model...
    return test_labels
```

CIFAR-10 Dataset

- CIFAR-10 Dataset
  - 10 class, 50000 training images, 10000 test images
  - Each image's size: 32 × 32 pixel

CIFAR-10 Dataset

Training Set

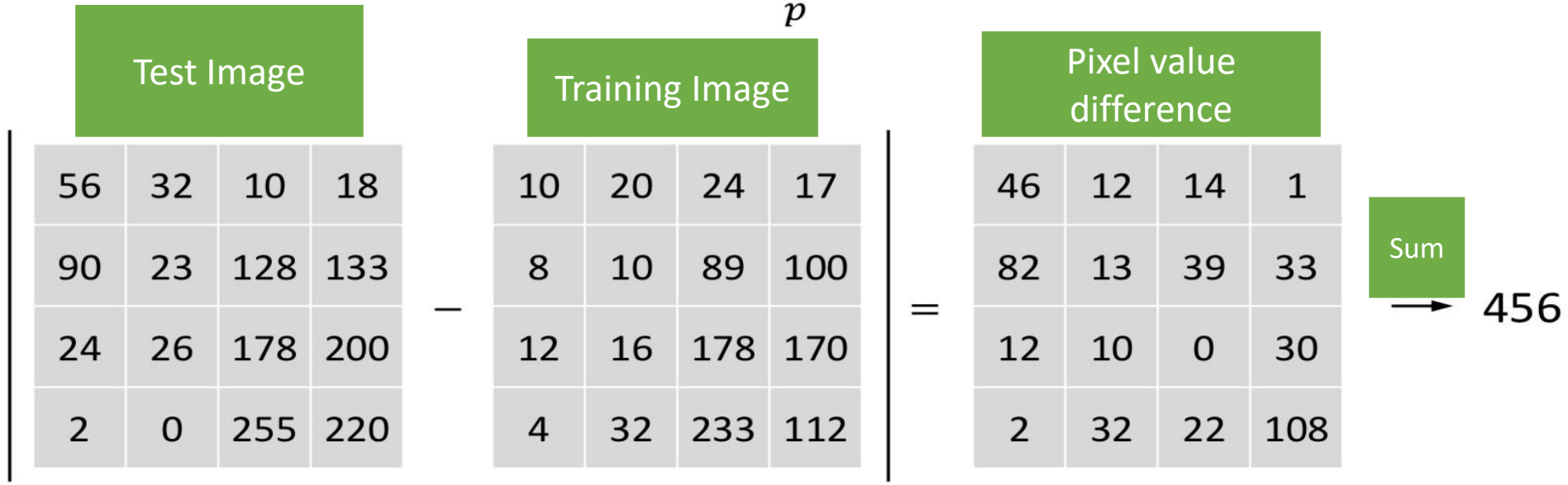


For each test image (first column), a number of its nearest neighbors are shown in the opposite row of that image.

Comparison of two images' similarity

- Distance measure.
  - L1 distance measure

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



Nearest Neighbor Classifier: Training

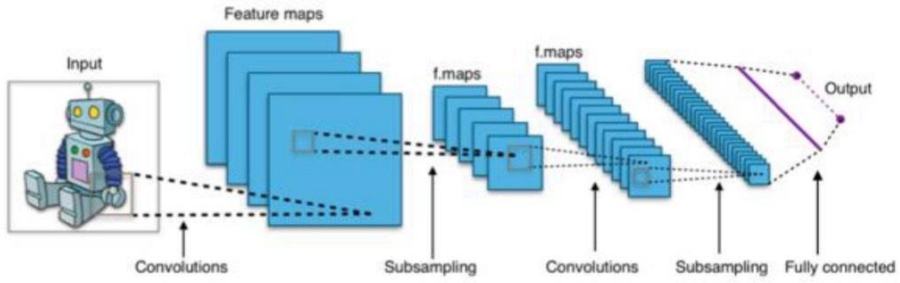
```python
import numpy as np

Class NearestNeighbor:

    def __init__(self):
        pass

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

    ...
```

Nearest Neighbor Classifier: Predict

```
def predict(self, X):
    """ X is N × D where each row is an example we wish to predict label for """
    num_test = X.shape[0]
    # make sure that 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 L1 distance
        distances = np.sum(np.abs(self.Xtr - X[i, :]), axis=1)
        min_index = np.argmin(distances)
        Ypred[i] = self.ytr[min_index]

    return Ypred
```

Nearest Neighbor Classifier

- Question: In classification using the nearest neighbor method, how does increasing the training set's size affect the classification speed (prediction)?
- Answer: Linearly!
- Attention: In practice, the speed of execution of a class in the prediction time is much more important than the speed of its execution in the learning time.
  - In the case of the nearest neighbor category, it is exactly the opposite!
- Deep Learning:
  - Learning(train) too time-taking
  - Predict(test) too fast



Distance measure

- Distance measure
  - Selecting distance measure, in nearest neighbor classification is a hyperparameter.
  - Common selections.

Manhattan distance(L1)

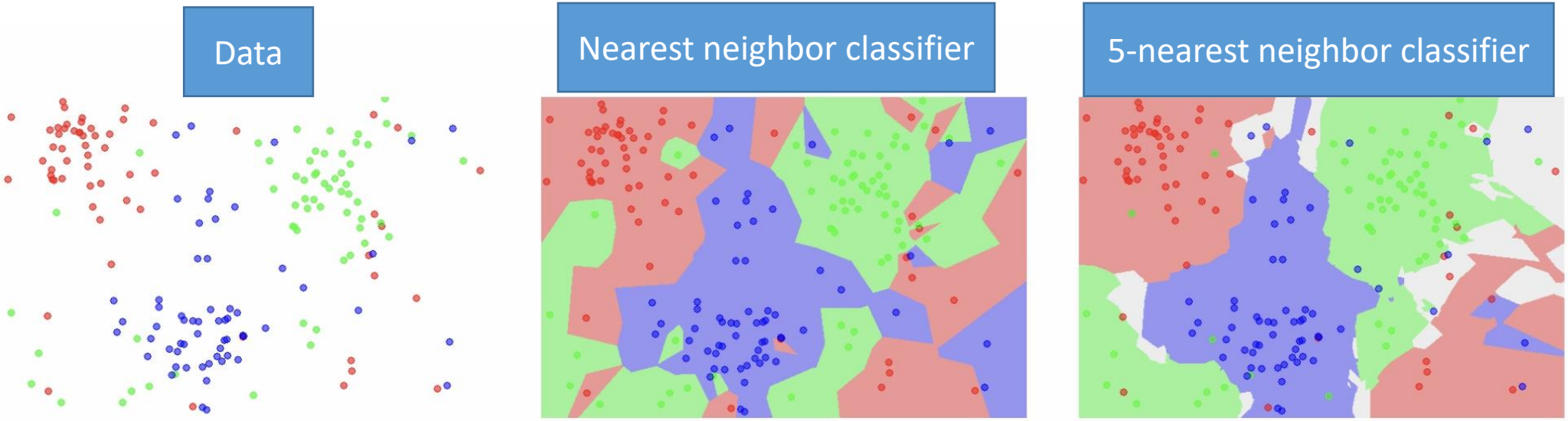
$$d_1(I_1, I_2) = \sum_p |I_1^p - I_2^p|$$

Euclidean distance(L2)

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

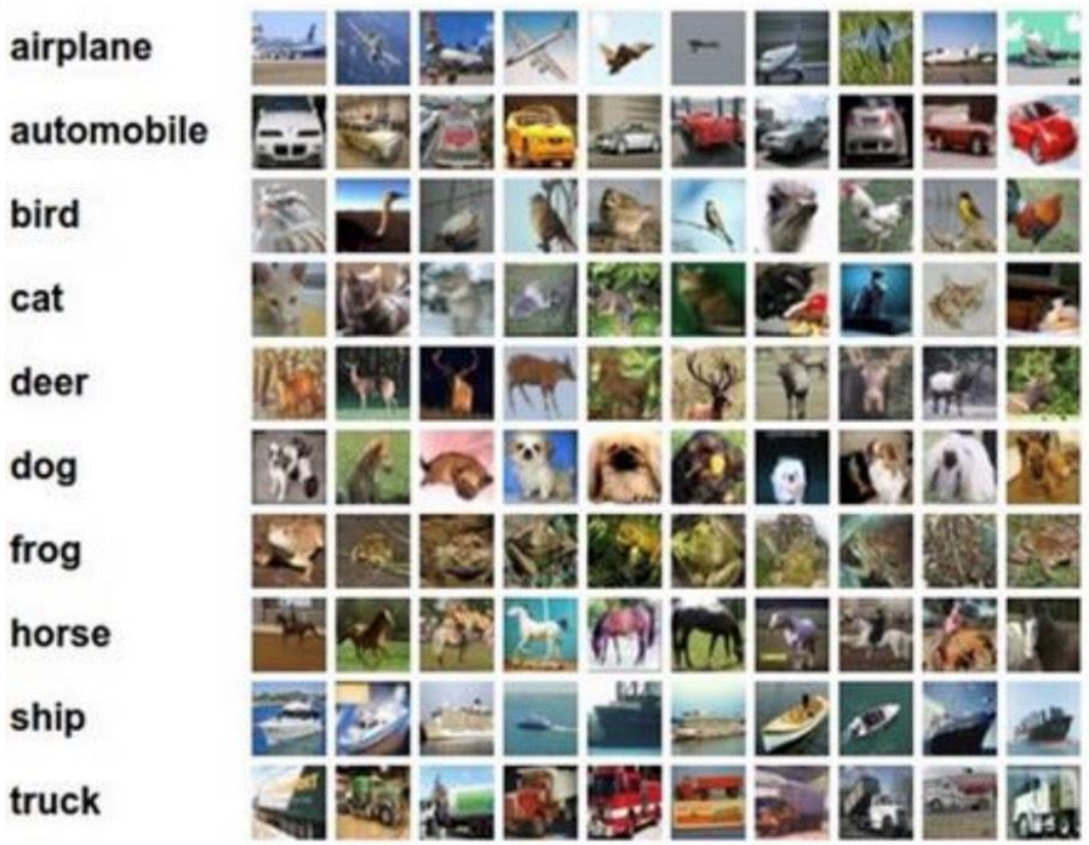
K-Nearest Neighbor Method

- K-Nearest Neighbor Method
  - At first, find the k nearer neighbors.
  - Then vote among them.

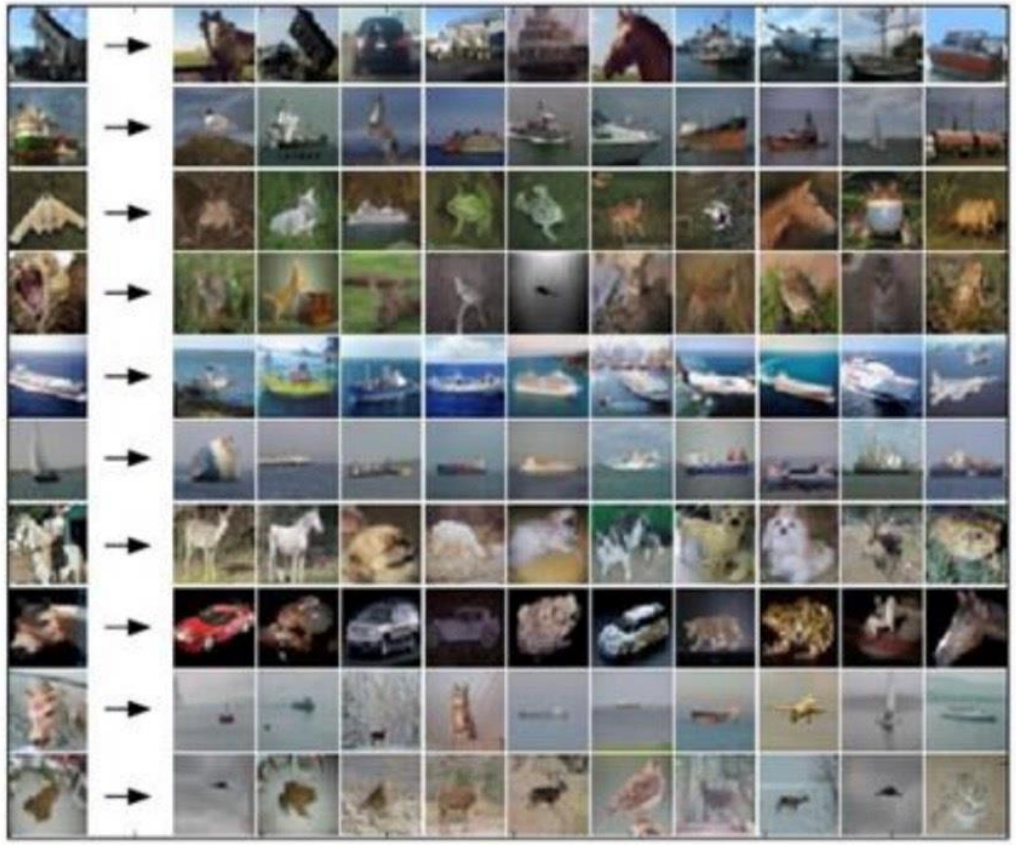


K-Nearest Neighbor Method

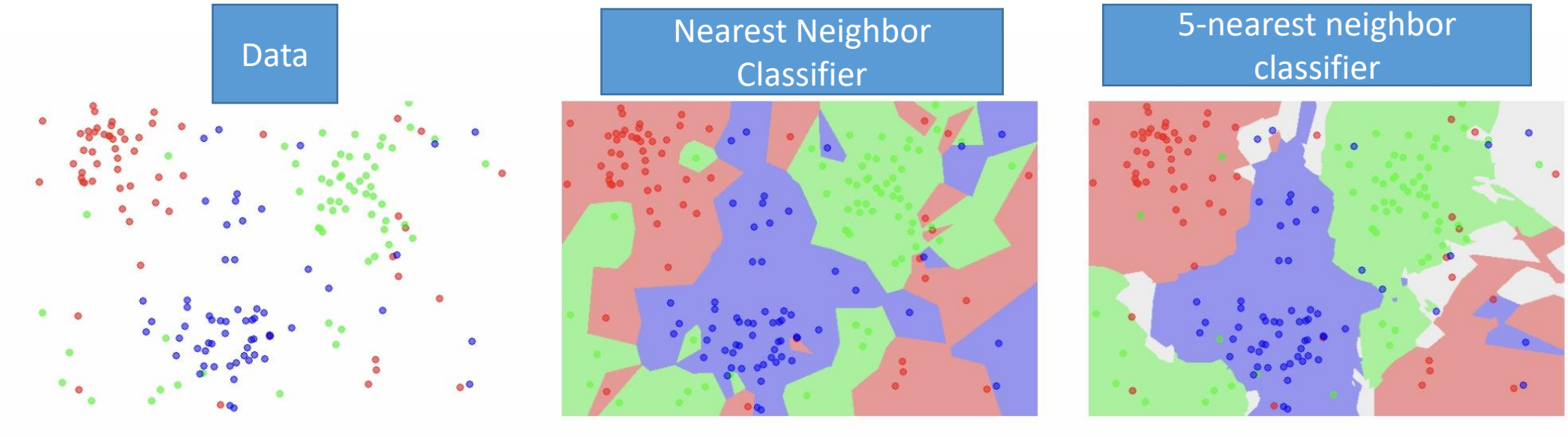
CIFAR-10 Dataset



For each test image (first column), a number of its nearest neighbors are shown in the opposite row of that image.

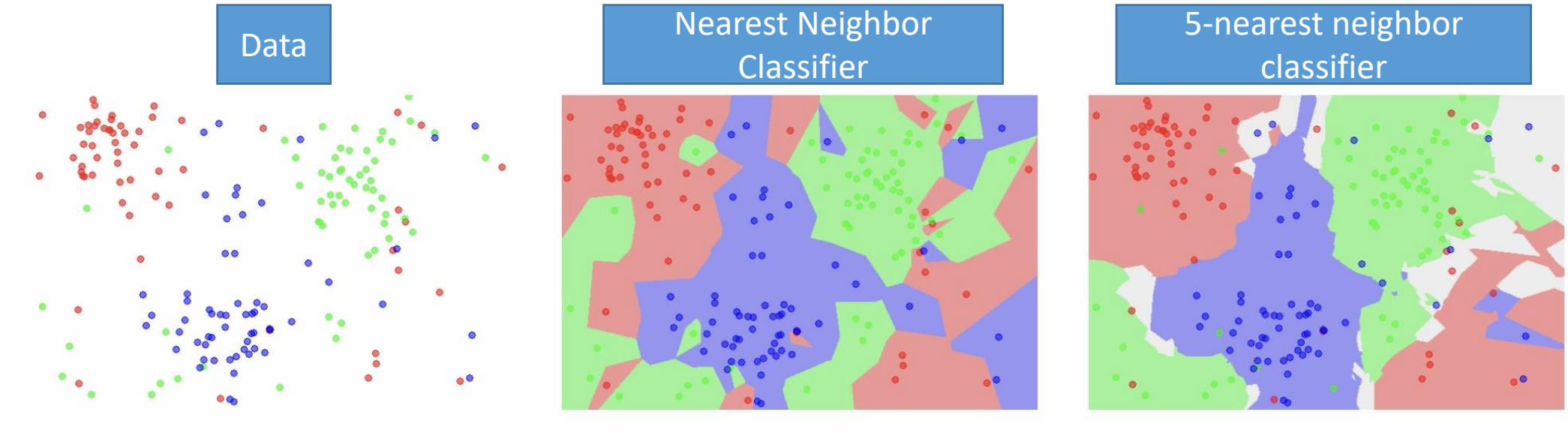


K-Nearest Neighbor Method



- Question: Using the Euclidean distance criterion, what is the accuracy of the nearest neighbor method on the training set?

K-Nearest Neighbor Method



- Question: Using the Euclidean distance criterion, what is the accuracy of the k-nearest neighbor method on the training set?

hyper parameters

- Selecting the best possible value for hyperparameters.
  - Which one is the best distance measure?
  - What is the best value for k?

- Selecting the best possible value for hyperparameters is completely based on the case.
  - At first, we should test all of the possible values for each hyperparameter!
  - And then choose the best one.

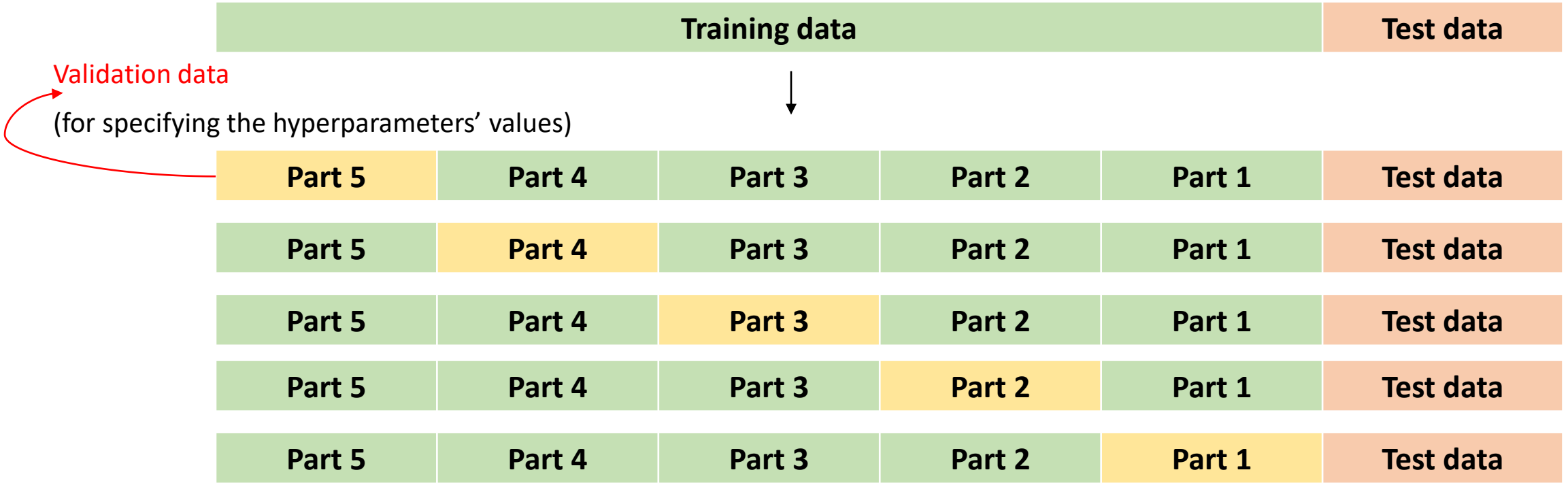
hyper parameters

- First solution (a very bad idea):
  - Choosing the value which leads to the highest classifier's accuracy on the test set.

- Attention (very important):
  - The test set is used at the end of the steps, and only to estimate the generalizability of the classifier.

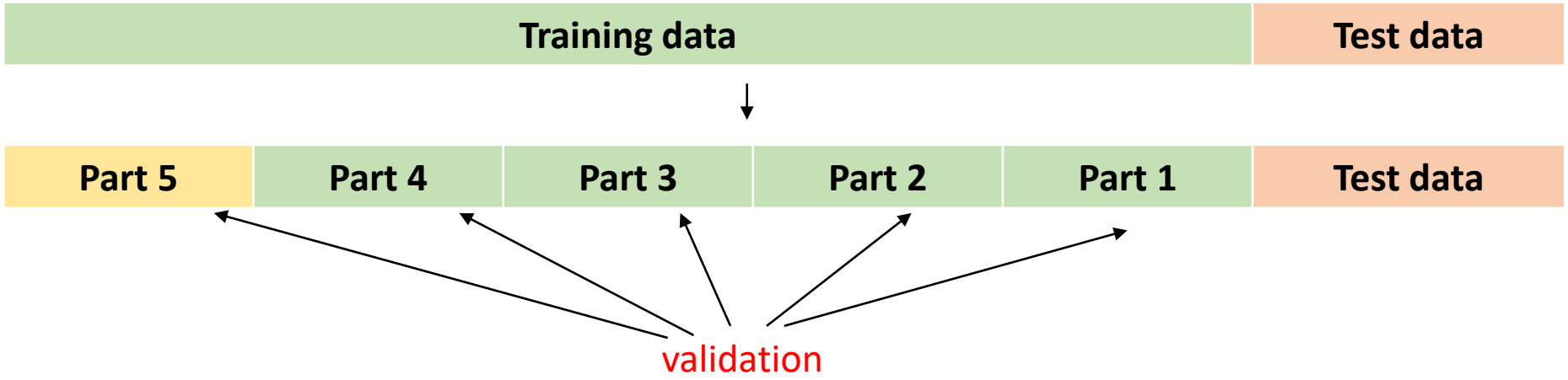
hyper parameters

- Second solution (Multipartite validation)



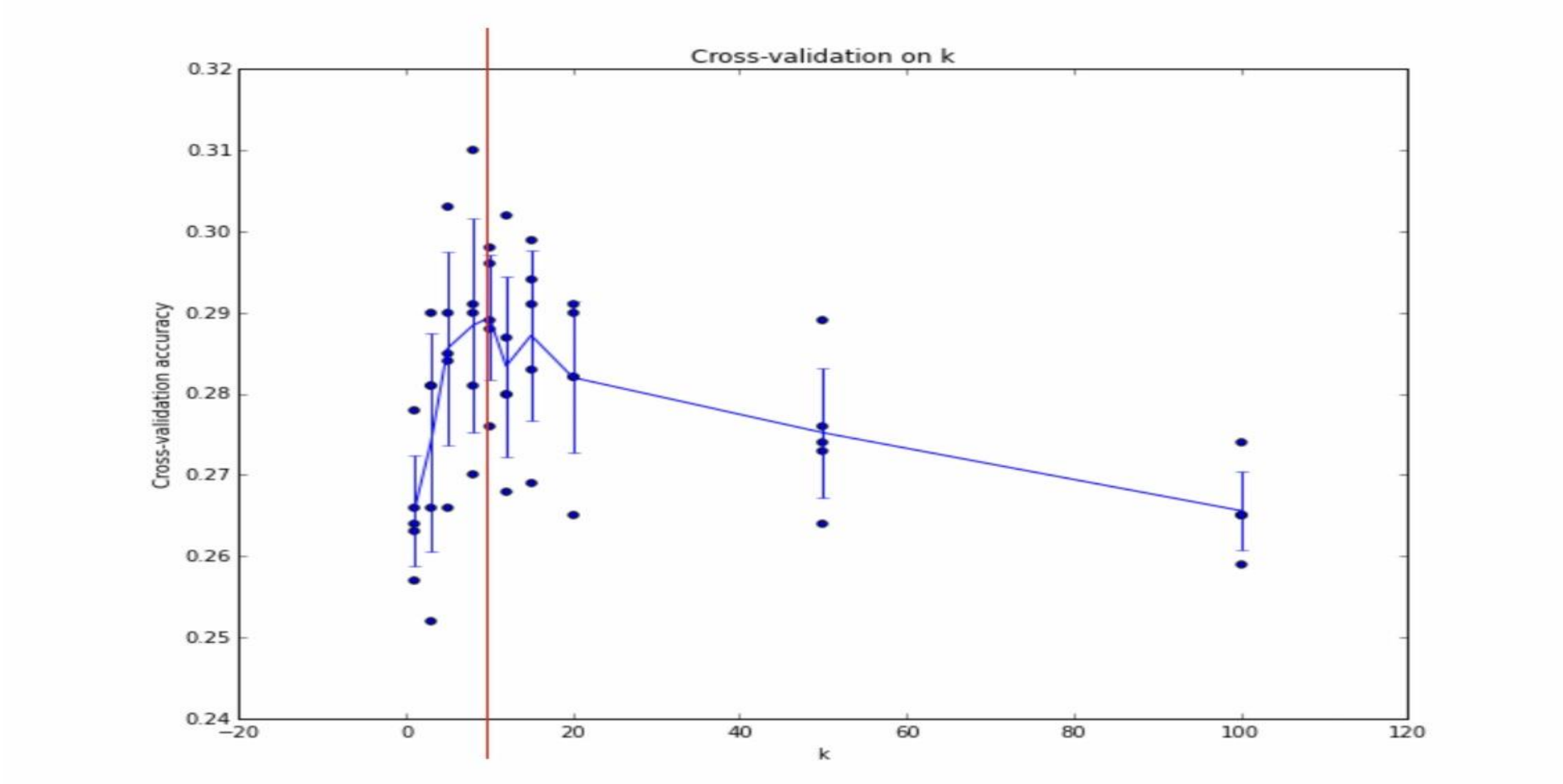
hyper parameters

- Second solution (Multipartite validation)



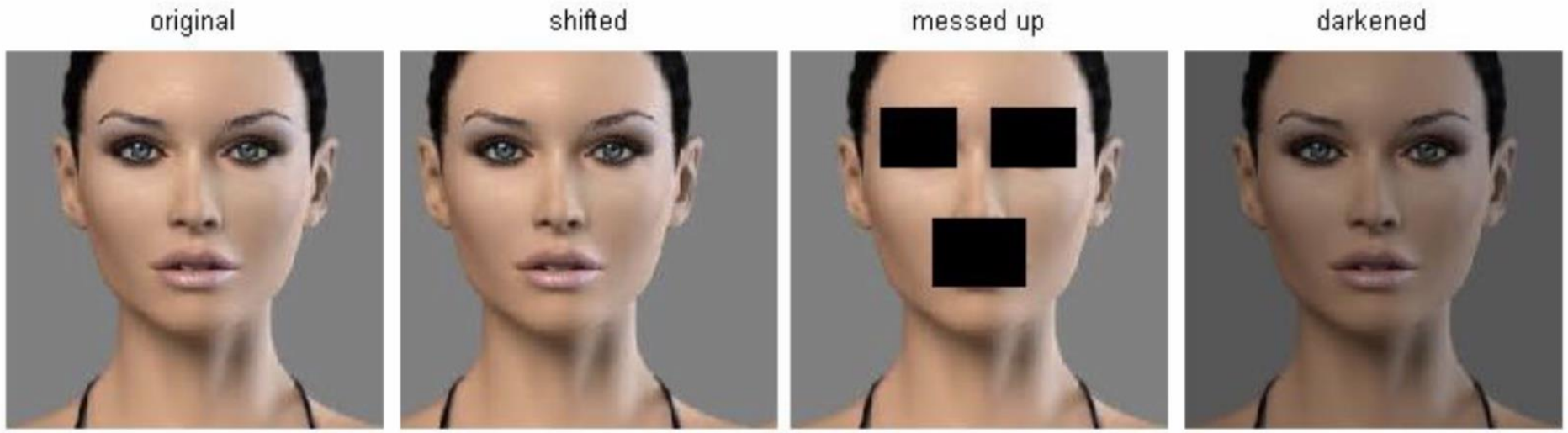
Each time, select one segment as validation data and then average the results.

Multipartite Validation



K-Nearest Neighbor Method and Image Classification

- Attention: Never use K-nearest neighbor for image classification.
  - Very slow in predicting (test)
  - The inappropriateness of using distance criteria at the level of the entire image!



All three transformed images have the same Euclidean distance with the original image (left image).

Conclusion

- Image Classification
  - Training set and test set
- Classification with the K-Nearest Neighbor method
  - Manhattan and Euclidean distance measures to calculate the similarity between two images
- Specifying the appropriate value for hyperparameters.
  - Validation set
  - Multipartite Validation
- Estimating the estimation ability of the classifier using the test set.

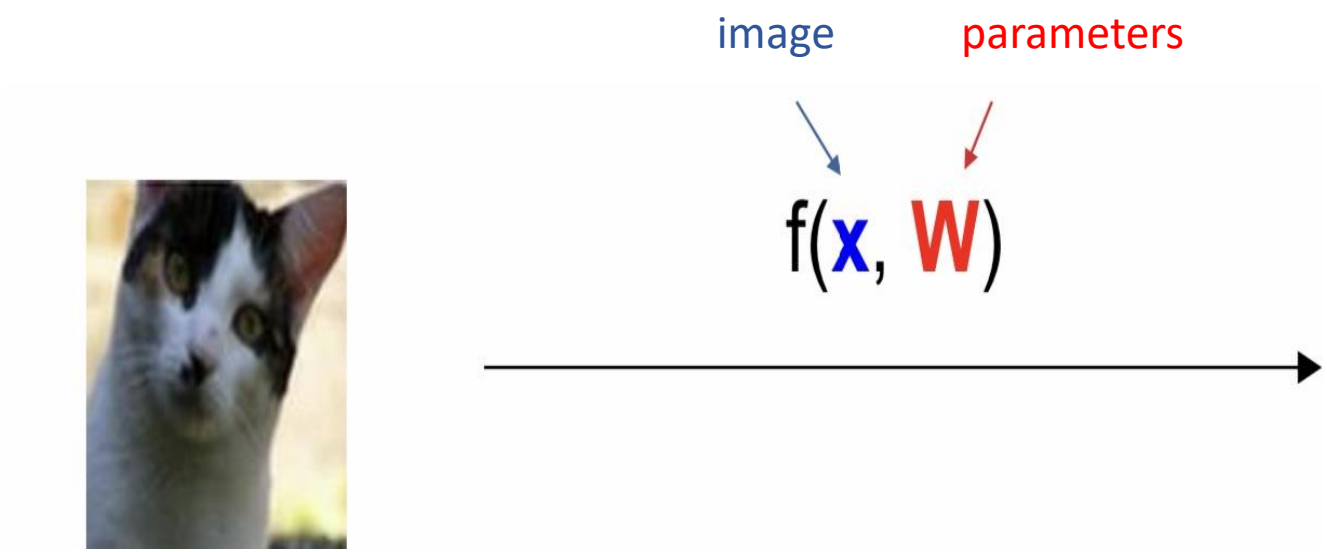
Linear Classification

Table of Content

- Linear classification
- Parametric and non-parametric methods
- Cost function (loss function)

Parametric approach: Linear Classification

- Linear classification



10 numbers
Each number represents the score of a class.

[32 × 32 × 3]
A vector from numbers in [0, 1]

Parametric approach: Linear Classification

- Linear classification

parameters, or “weights”

10 × 1 10 × 3072 3072 × 1 10 × 1

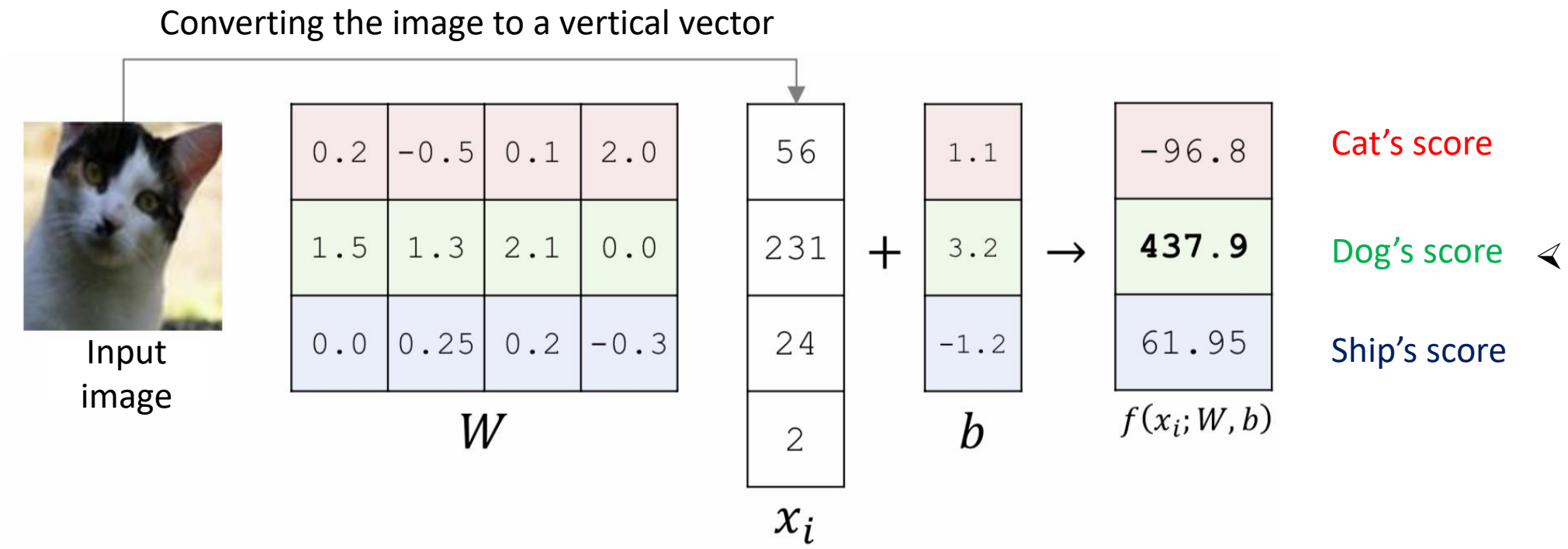
$$f(x, W) = W\,x + (+b)$$

10 numbers
Each number represents the score of a class.

[32 × 32 × 3]
A vector from numbers in [0, 1]

Linear Classification: Example

- Four features and three classes (dog, cat, ship)

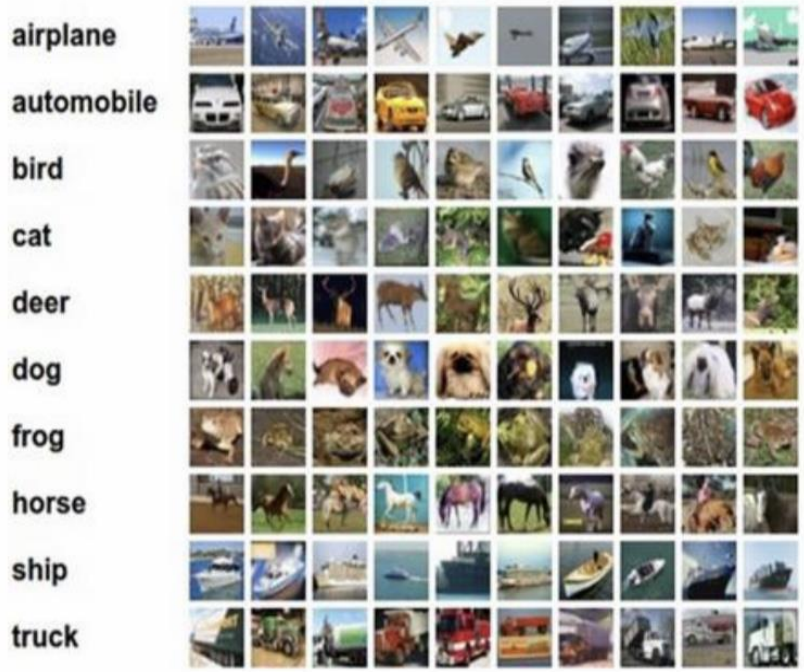


Linear Classification Interpretation

- What operation does a linear classification?

$$f(x_i; W, b) = Wx_i + b$$

Linear Classification Interpretation

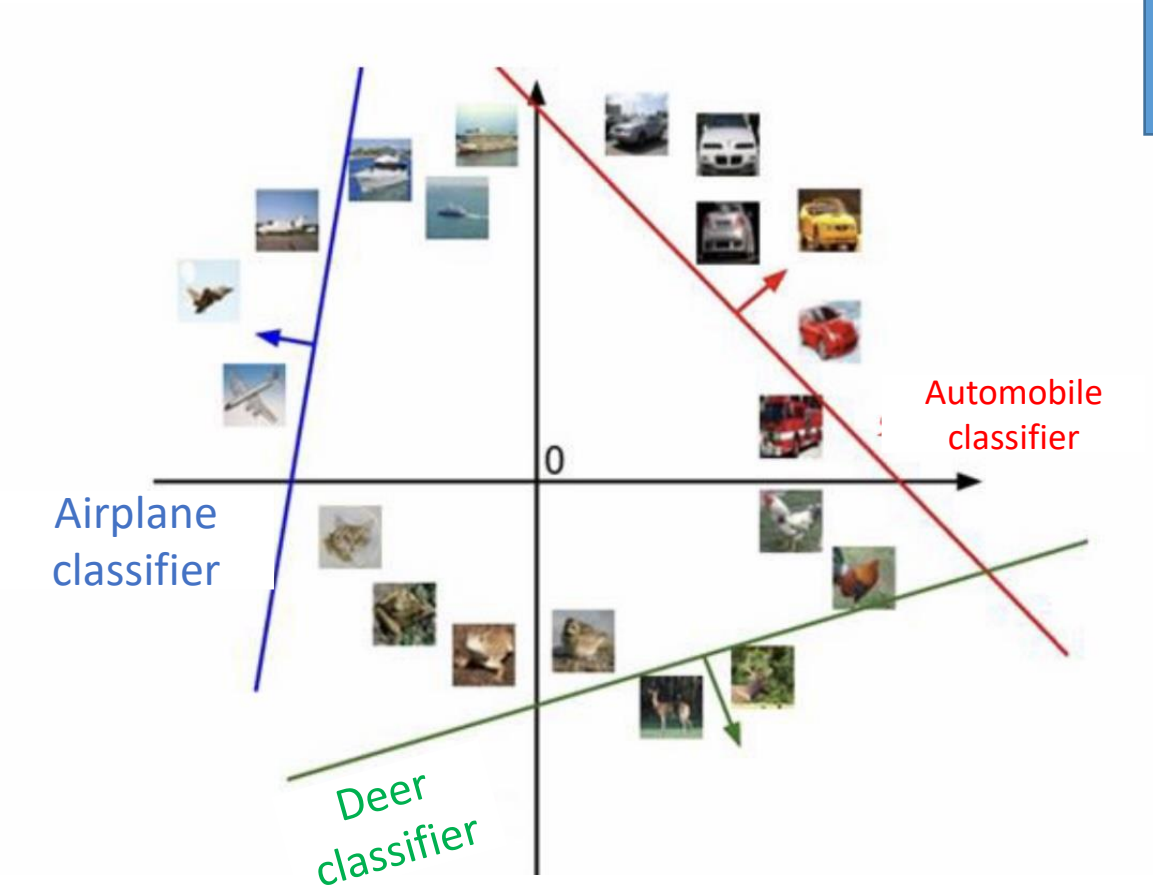


What operation does a linear classification?

$$f(x_i; W, b) = Wx_i + b$$

An example of the weights trained using a linear classifier on the CIIFAR-10 set.

Linear Classification Interpretation



What operation does a linear classification?

$$f(x_i; W, b) = Wx_i + b$$

[32 × 32 × 3]
A vector from numbers in [0, 1]

Linear Classification: Score Function

Score different classes for three input images using random weights.



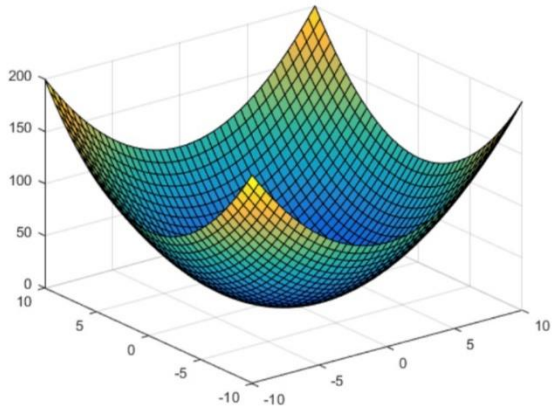
-3.45	-0.51	3.42	Airplane
-8.87	**6.04**	4.64	Automobile
0.09	5.31	2.65	Bird
2.90	-4.22	5.10	Cat
4.48	-4.19	2.64	deer
8.02	3.58	5.55	Dog
3.78	4.49	**-4.34**	Frog
1.06	-4.37	-1.50	Horse
-0.36	-2.09	-4.79	Ship
-0.72	-2.93	6.14	Truck

$$f(x_i; W, b) = Wx_i + b$$

[32 × 32 × 3]
A vector from numbers in [0, 1]

In the following …

- Cost function (loss function)
  - Expressing the badness of the weight matrix as a numerical value!
- Optimization
  - Finding a weight matrix that minimizes the cost function.
- Non-linear classification.



Cost Function

Cost Function and Optimization

- Cost function definition
  - in a way that expresses our dissatisfaction with the points calculated on the data of the training set.
- Optimization
  - Finding the parameter values so that the cost function is minimized.

Airplane	-3.45	-0.51	3.42
Automobile	-8.87	**6.04**	4.64
Bird	0.09	5.31	2.65
Cat	**2.90**	-4.22	5.10
deer	4.48	-4.19	2.64
Dog	**8.02**	3.58	5.55
Frog	3.78	4.49	**-4.34**
Horse	1.06	-4.37	-1.50
Ship	-0.36	-2.09	-4.79
Truck	-0.72	-2.93	**6.14**

Cost Function and Optimization



Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

- Training set
  - Including 3 training instances from 3 different classes
- Score function
  - Calculating the score of each class for each data

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

Multiclass SVM cost function



Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**

Calculating the cost of each data(x,y):

Input image $:x_i$

The label of the input image (an integer) $:y_i$

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

Multiclass SVM cost function

Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9		

Calculating the cost of each data(x,y):

Input image x_i

The label of the input image (an integer) y_i

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

```
= max(0,  5.1 - 3.2 + 1) +
  max(0, -1.7 - 3.2 + 1)
= max(0, 2.9) + max(0, -3.9)
= 2.9 + 0
= 2.9
```

Multiclass SVM cost function

Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	**0.0**	

Calculating the cost of each data(x,y):

Input image $:x_i$

The label of the input image (an integer) $:y_i$

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

```
= max(0, 1.3 - 4.9 + 1) +
  max(0, 2.0 - 4.9 + 1)
= max(0, -2.6) + max(0, -1.9)
= 0 + 0
= 0
```

Multiclass SVM cost function



Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	**0.0**	**12.9**

Calculating the cost of each data(x,y):

Input image $:x_i$

The label of the input image (an integer) $:y_i$

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

```
= max(0, 2.2 - (-3.1) + 1) +
  max(0, 2.5 - (-3.1) + 1)
= max(0, 6.3) + max(0, 6.6)
= 6.3 + 6.6
= 12.9
```

Multiclass SVM cost function



Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	**0.0**	**12.9**

L = (2.9 + 0 + 12.9) / 3 = 5.27

Calculating the cost of each data(x,y):

- Input image x_i
- The label of the input image (an integer) y_i

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

Total cost:

- Average cost over all training data

$$L = \frac{1}{N} \sum_i L_i$$

Multiclass SVM cost function



Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0.0	12.9

Calculating the cost of each data(x,y):

Input image x_i

The label of the input image (an integer) y_i

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

If the total is calculated on all classes, how does the cost function change?

Multiclass SVM cost function



Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0.0	12.9

Calculating the cost of each data(x,y):

Input image $:x_i$

The label of the input image (an integer) $:y_i$

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

If the average is used instead of the total, how does the cost function change?

Multiclass SVM cost function

Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0.0	12.9

Calculating the cost of each data(x,y):

Input image x_i

The label of the input image (an integer) y_i

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

If the following relationship is used instead of this relationship, how does the cost function change?

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)^2$$

Multiclass SVM cost function

Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0.0	12.9

Calculating the cost of each data(x,y):

Input image $:x_i$

The label of the input image (an integer) $:y_i$

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

What is the minimum and maximum possible value for the cost function?

Multiclass SVM cost function

Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	0.0	12.9

Calculating the cost of each data(x,y):

Input image x_i

The label of the input image (an integer) y_i

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

Normally, at the starting moment, the values of w are initialized with small random values and therefore all scores are almost zero. In this case, what is the value of the cost function?

Multiclass SVM cost function

- Implementation in Python

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

```
def L_i_vectorized(x, y, W):
    scores = W.dot(x)
    margins = np.maximum(0, scores - scores[y] + 1)
    margins[y] = 0
    loss_i = np.sum(margins)
    return loss_i
```

There is a problem in the cost function

- Suppose we find a W such that L = 0. Is W unique?

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

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1)$$

There is a problem in the cost function



Cat	**3.2**	1.3	2.2
Automobile	5.1	**4.9**	2.5
frog	-1.7	2.0	**-3.1**
	2.9	**0.0**	

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

Before:

```
= max(0, 1.3 - 4.9 + 1) +
  max(0, 2.0 - 4.9 + 1)
= max(0, -2.6) + max(0, -1.9)
= 0 + 0
= 0
```

If we double the value of W.

```
= max(0, 2.6 - 9.8 + 1) +
  max(0, 4.0 - 9.8 + 1)
= max(0, -6.2) + max(0, -4.8)
= 0 + 0
= 0
```

Solution: regularize the weights

Regularize the weights:

Regularization factor

$$L = \frac{1}{N}\sum_{i=1}^{N}\sum_{j\neq y_i}\max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

Some Common Methods:

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

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

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

Soft Max Classifier (Multi class Logistic Regression)

Scores. The logarithm of the probability of classes is not normalized!

Cat 3.2

Automobile 5.1

frog -1.7

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

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

Soft Max function

Target:
Maximization of the logarithm of the right exponent (or minimization of the negative logarithm of the right exponent)!

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

Soft Max Classifier (Multi class Logistic Regression)

Scores. The logarithm of the probability of classes is not normalized!

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

$$L_i = -\log(0.13) = \mathbf{0.89}$$

Cat

Automobile

frog

	exponentiation		normalization	
3.2	→	24.5	→	0.13
5.1	→	164.0	→	0.87
-1.7	→	0.2	→	0.0

The non-normal logarithm of probabilities

The non-normal probabilities

The probabilities

What is the minimum and maximum possible value for the cost function?

Soft Max Classifier (Multi class Logistic Regression)

Scores. The logarithm of the probability of classes is not normalized!

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

$$L_i = -\log(0.13) = \mathbf{0.89}$$

Cat

Automobile

frog

	exponentiation		normalization	
3.2	→	24.5	→	0.13
5.1	→	164.0	→	0.87
-1.7	→	0.2	→	0.0

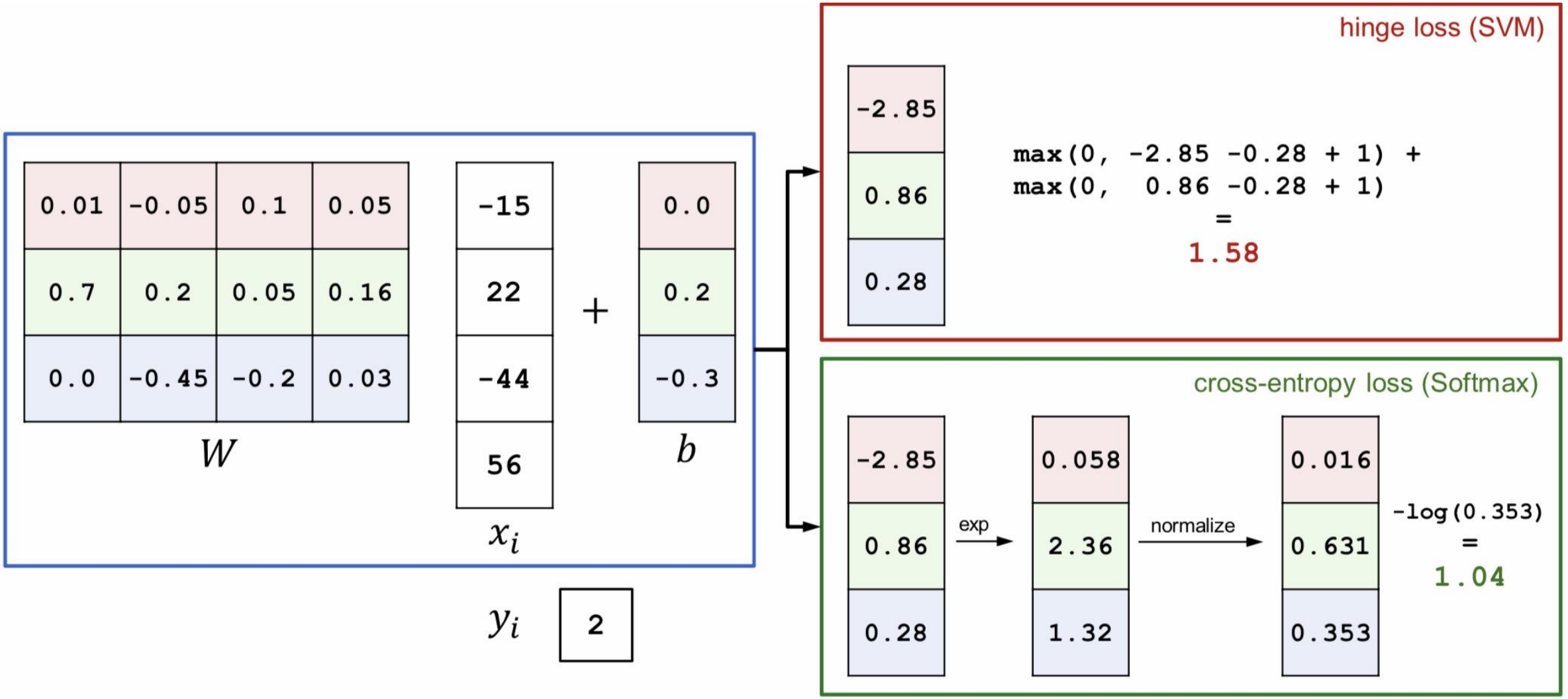
The non-normal logarithm of probabilities

The non-normal probabilities

The probabilities

Normally, at the starting moment, the values of w are initialized with small random values and therefore all scores are almost zero. In this case, what is the value of the cost function?

Cost Functions



Comparison

- Question: Suppose we take a data and move it slightly. (Let's change its scores a little.) How does the amount of cost change for each of the two given cost functions?

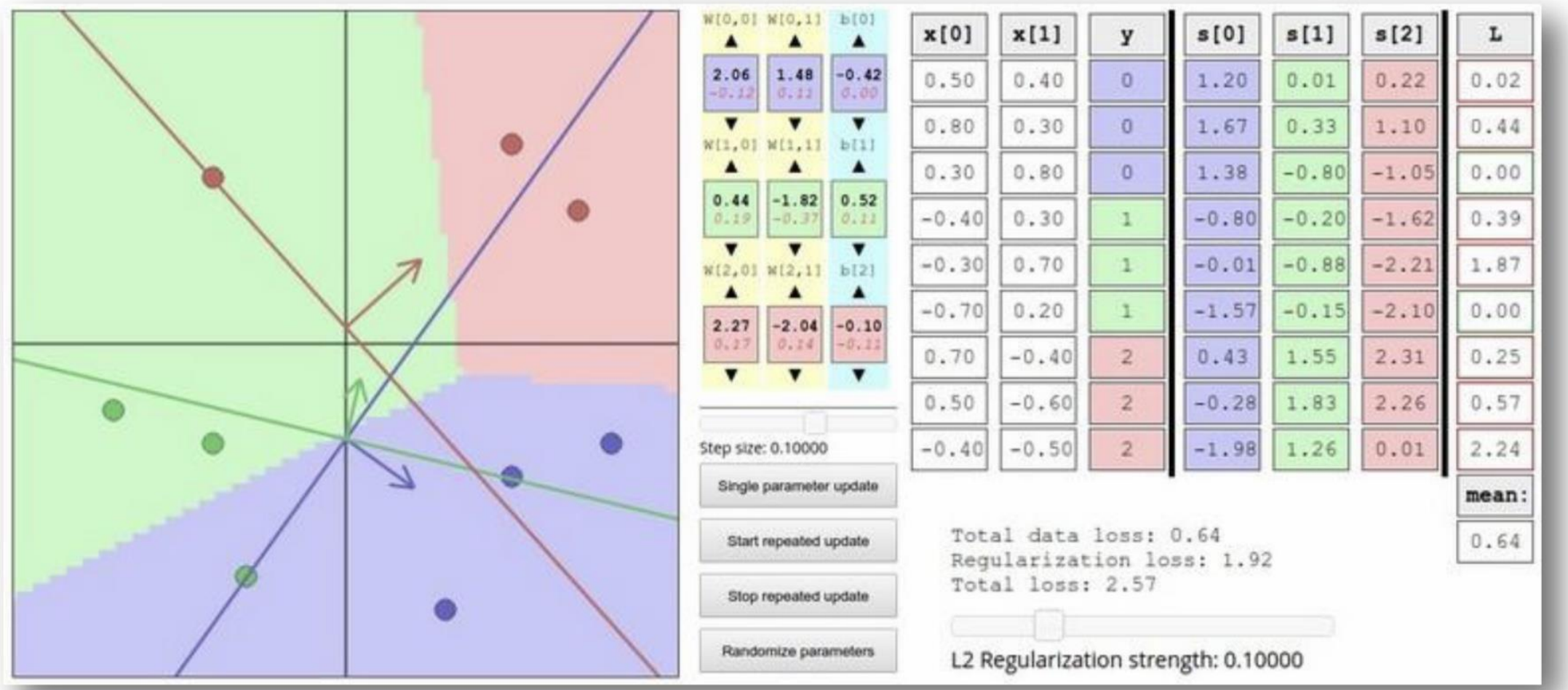
$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

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

$[y_i = 0]$ Scores

```
[10,   -2,    3]
[10,    9,    9]
[10, -100, -100]
```

Theatrical performance



http://vision.stanford.edu/teaching/cs231n/linear-classify-demo/

Optimization

Reminder

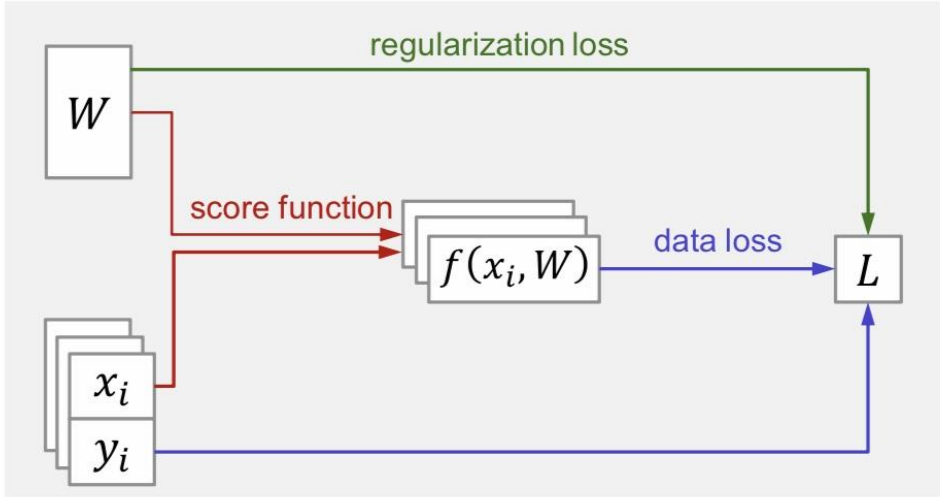
- A set of training data as (x,y)
- A score function: s = f(x;W) = Wx
- A cost function:

Target:
Minimize the cost function in order to find the values of W

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

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

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



First strategy: random search (a very bad idea)

```
best_loss = float("inf")
for i in xrange(1000):
    W = np.random.randn(10, 3073) * 0.0001
    loss = L(X_train, y_train, W)
    if loss < best_loss:
        best_loss = loss
        best_W = W
    print "In attempt %d the loss was %f, best %f" % (i, loss, best_loss)
```

```
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
```

An example of the implementation of the random search algorithm

Random search experiment

```
# assume X_test is (3073, 10000), y_test is (10000, 1)
scores = np.dot(best_W, X_test)
# find the index with max score in each column (the predicted class)
y_pred = np.argmax(scores, axis=0)
# calculate accuracy (fraction of predictions that are correct)
accuracy = np.mean(y_pred == y_test)
# returns 0.1555 as accuracy
```

- 15.5% accuracy! Not too bad!

(But the accuracy of the best solution is around 95 percent.)

Gradient Descent

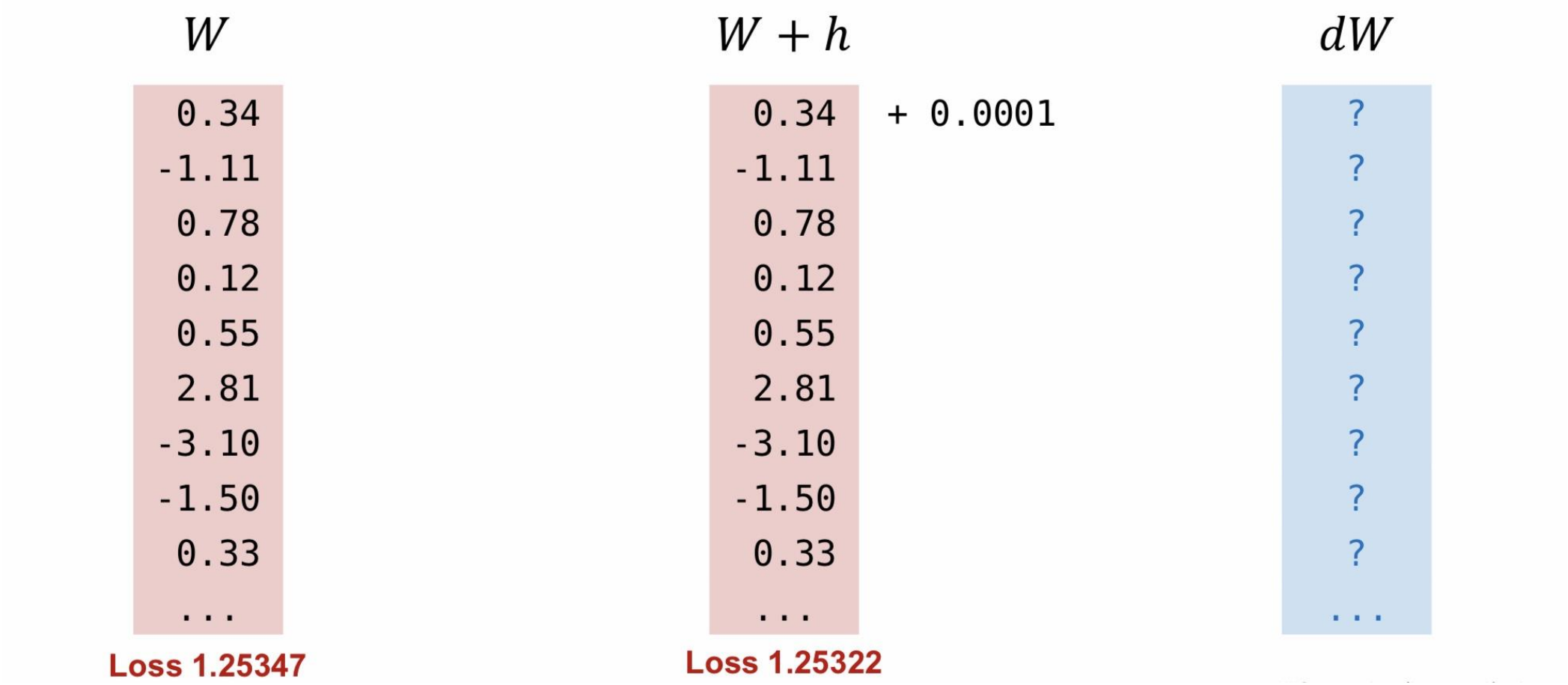
Second strategy: Follow the slope

- Differential function. in a 1-dimensional space

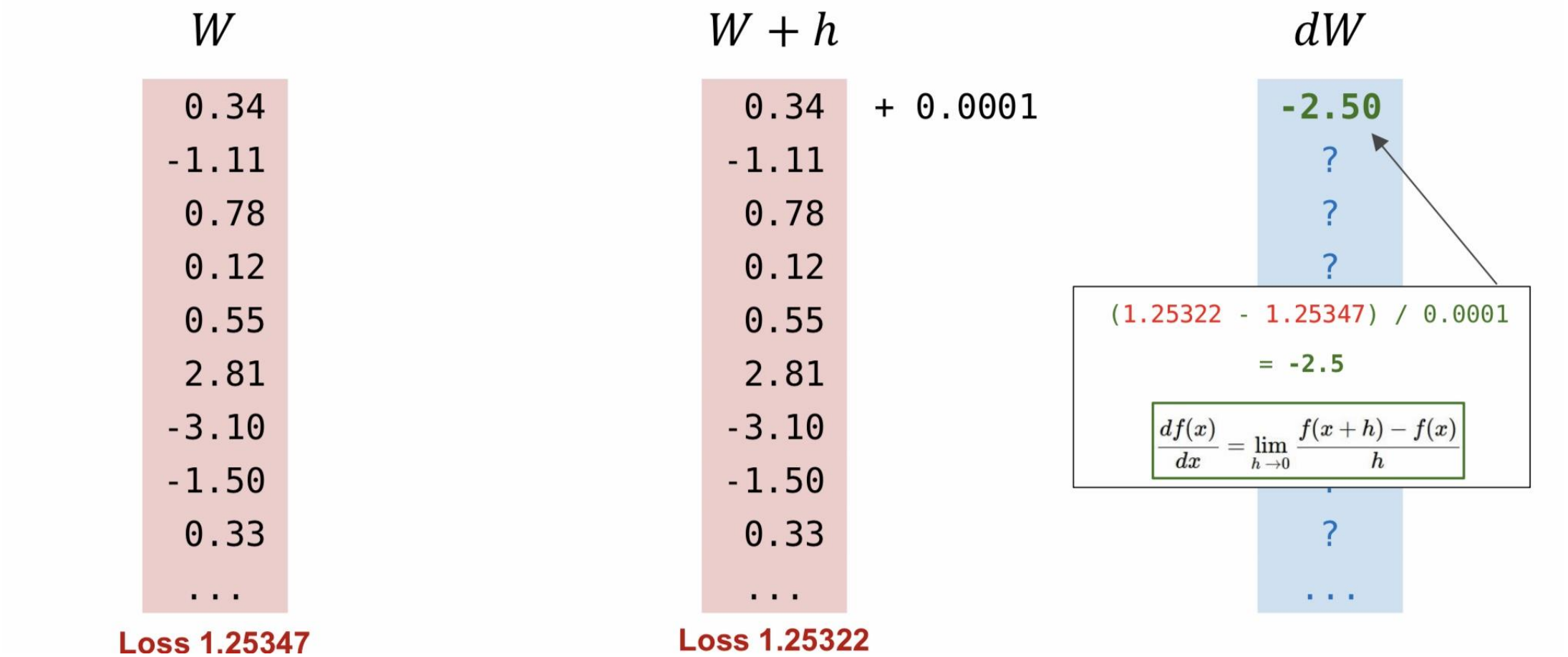
$$\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- In a multidimensional space, the gradient is a vector function of partial differentials

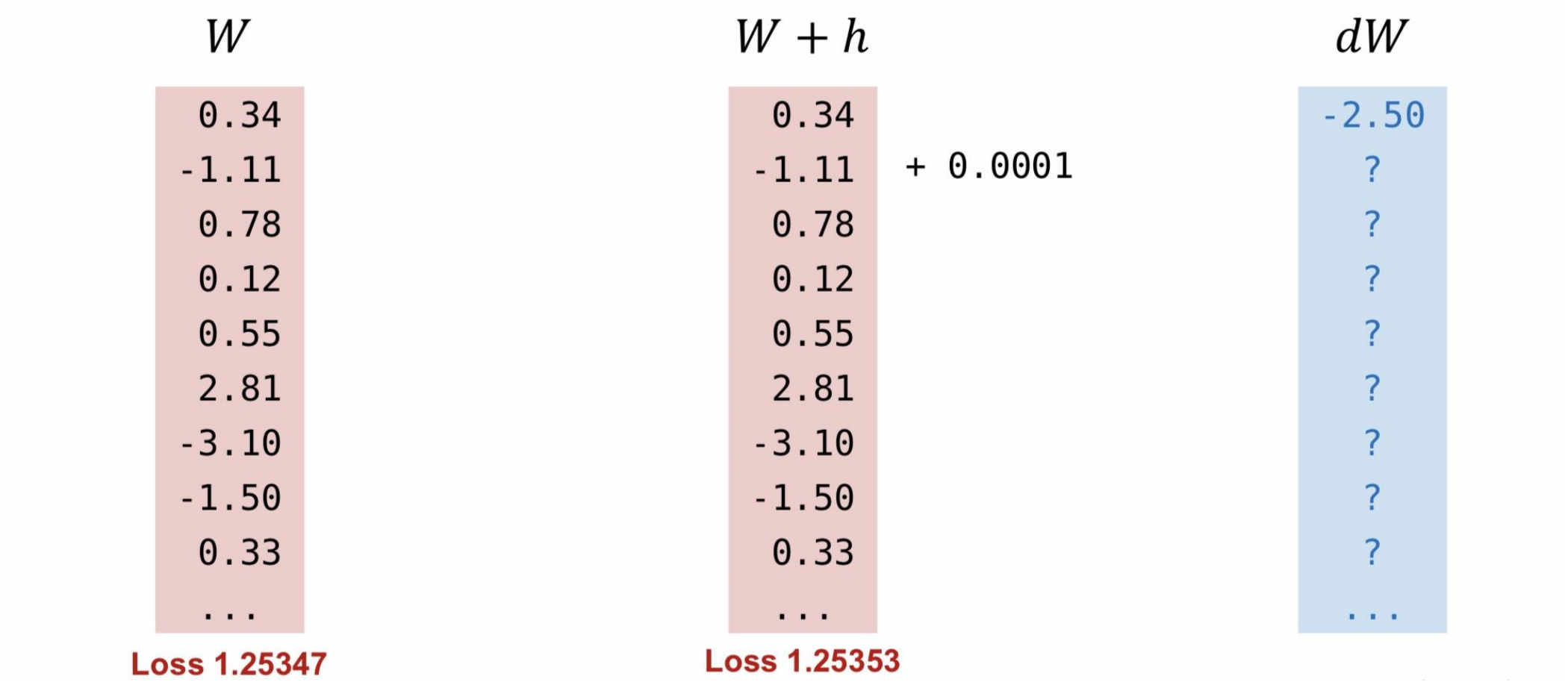
Numerical calculation of the gradient



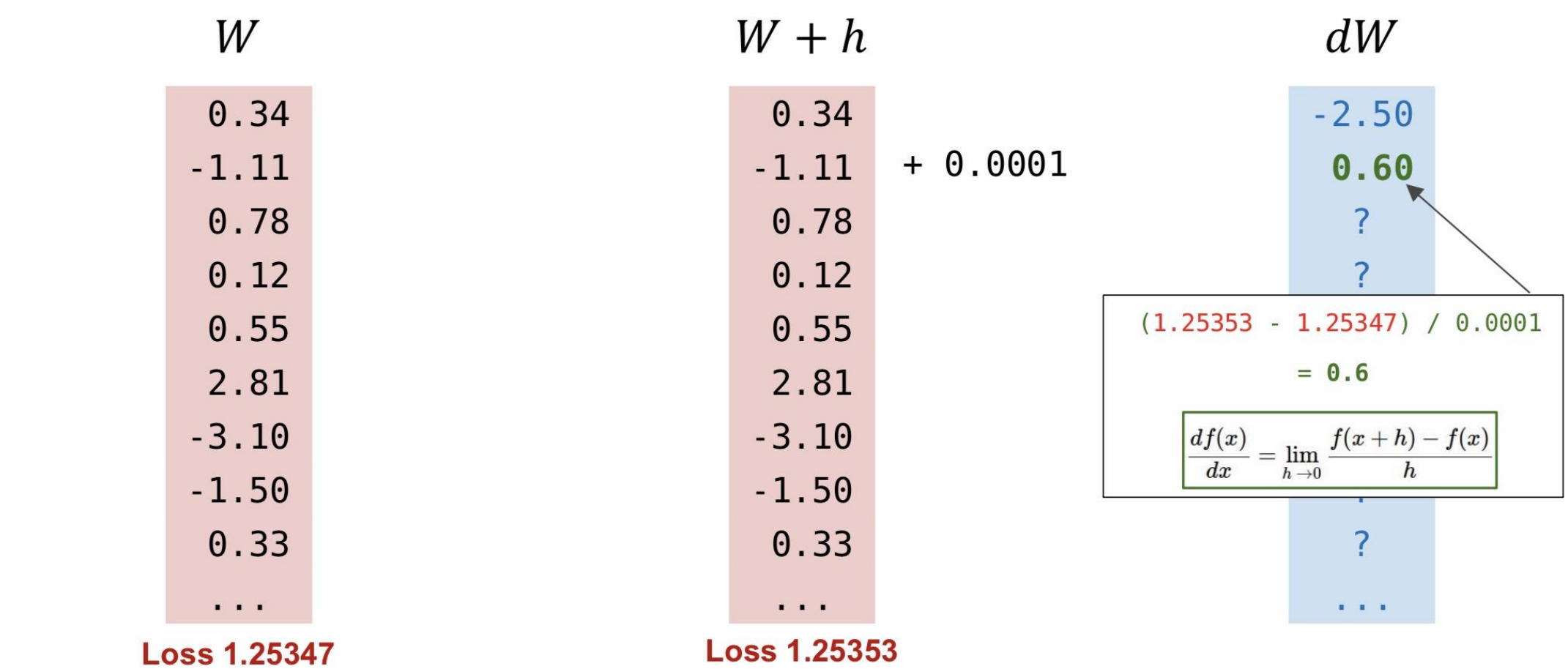
Numerical calculation of the gradient



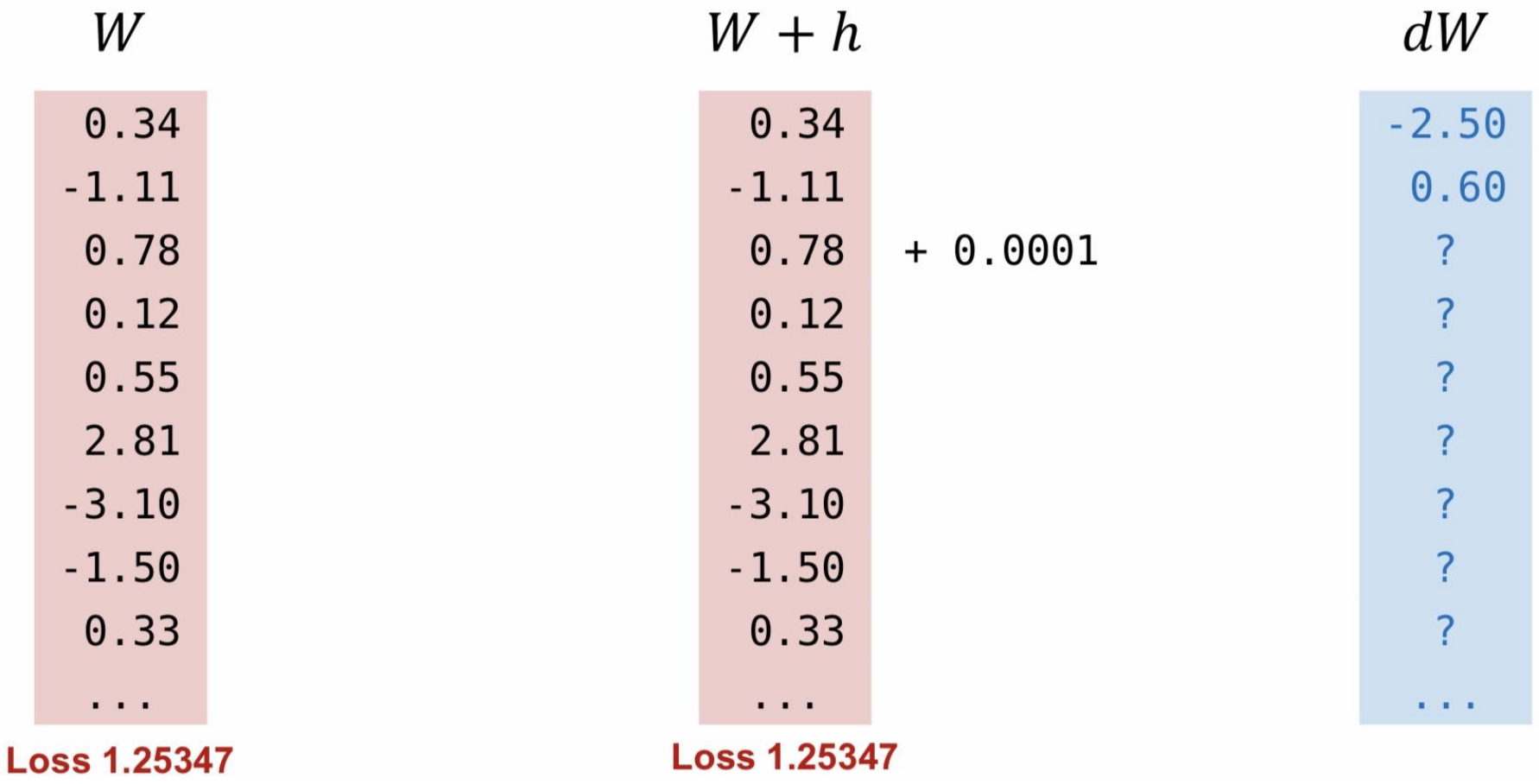
Numerical calculation of the gradient



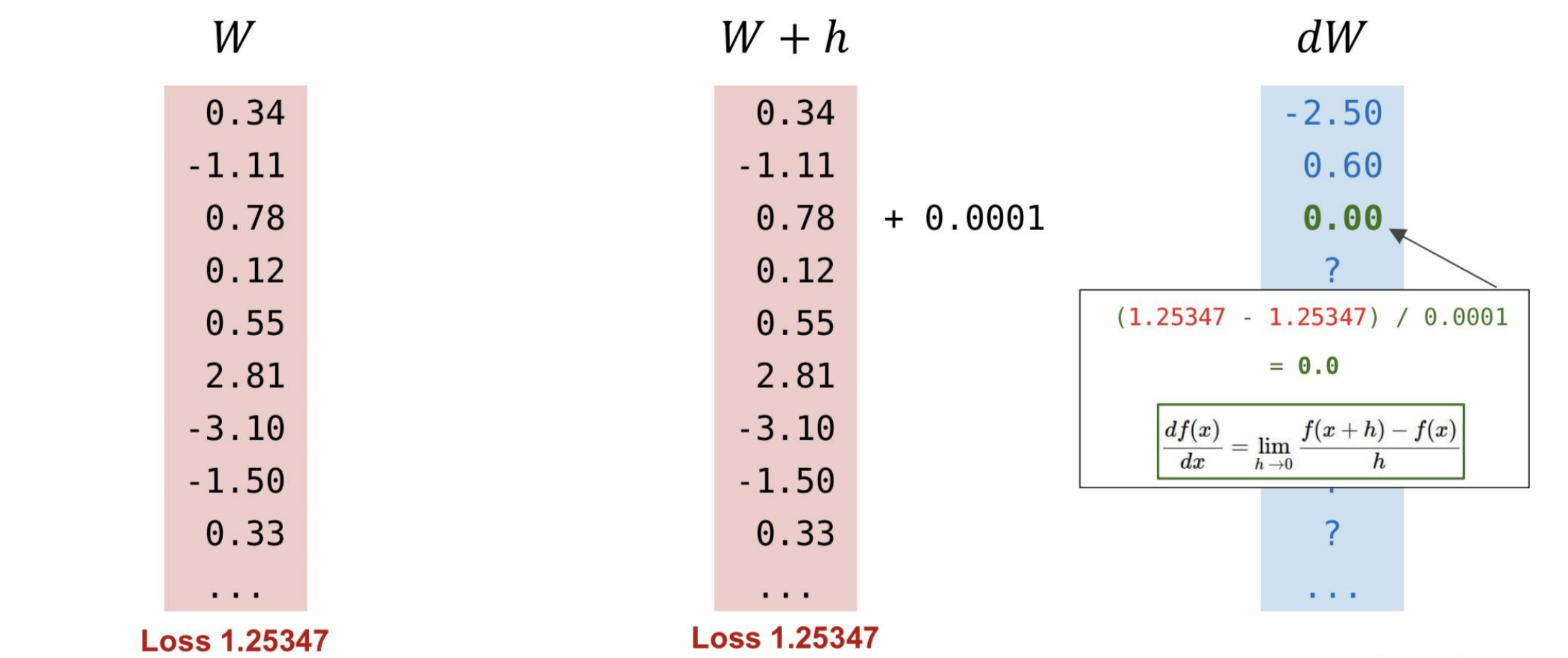
Numerical calculation of the gradient



Numerical calculation of the gradient



Numerical calculation of the gradient



Numerical calculation of the gradient

```
def eval_numerical_gradient(f, x):

    fx = f(x)
    grad = np.zeros(x.shape)
    h = 0.00001

    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:

        ix = it.multi_index
        old_value = x[ix]
        x[ix] += h
        fxh = f(x)  # evaluate f(x + h)
        x[ix] = old_value

        grad[ix] = (fxh - fx) / h  # compute the partial derivative
        it.iternext()  # step to next dimension

return grad
```

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

Disadvantages:

- Approximate
- very time consuming

Calculate the gradient analytically

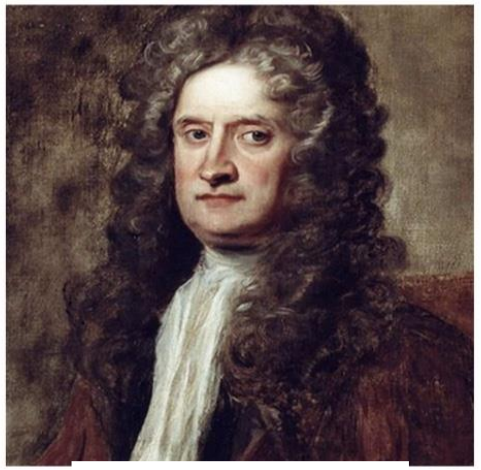
- The cost function is a function of W parameters.
  - Calculating the gradient numerically doesn't seem very smart.

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

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

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

$$\nabla_W L = \cdots$$



Isaac Newton
(1727-1642)

Wilhelm Leibniz
(1716-1642)

Calculate the gradient analytically

- The cost function is a function of W parameters.
  - Calculating the gradient numerically doesn't seem very smart.

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

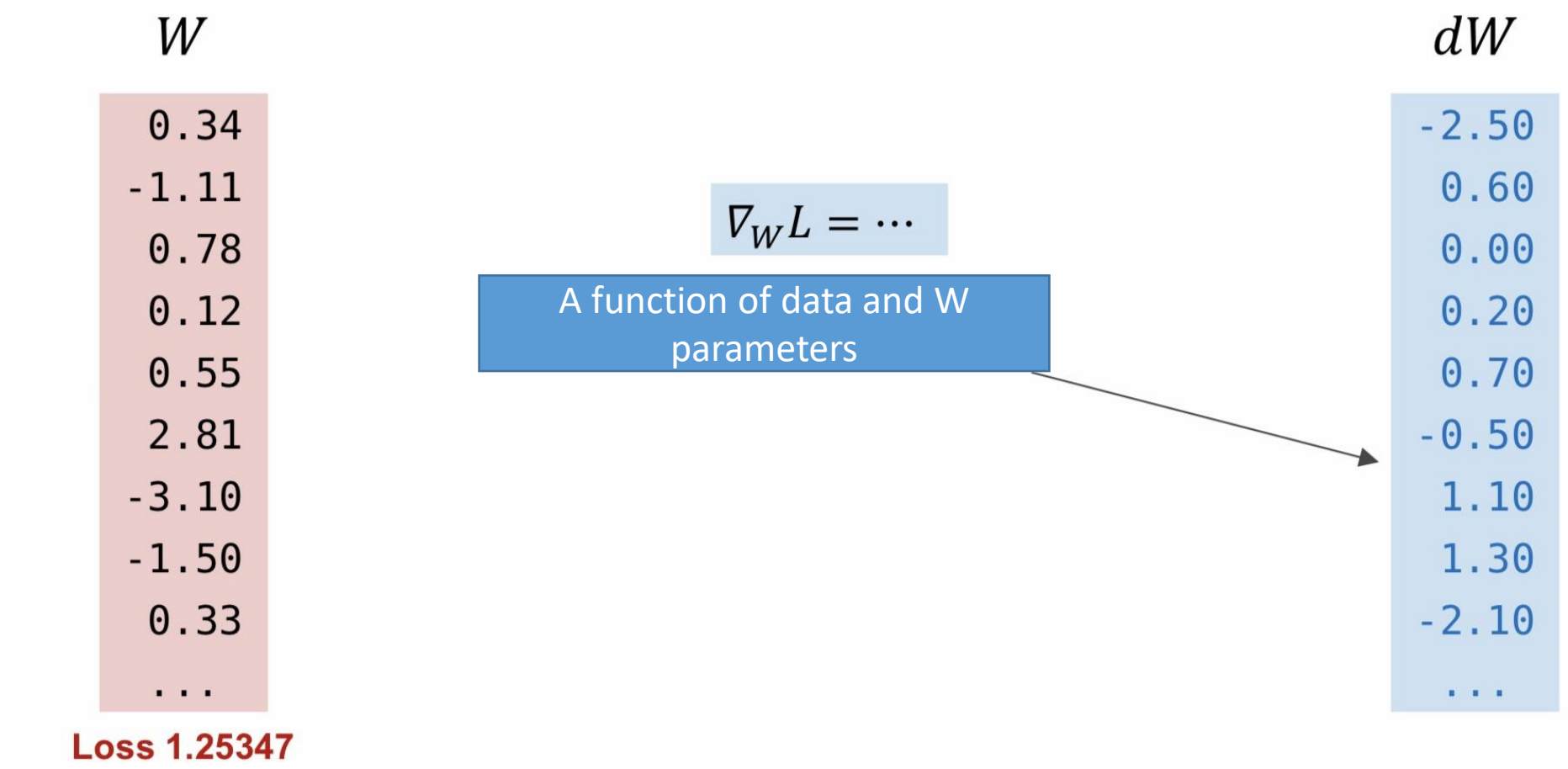
$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

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

$$\nabla_W L = \cdots$$

Numerical calculation

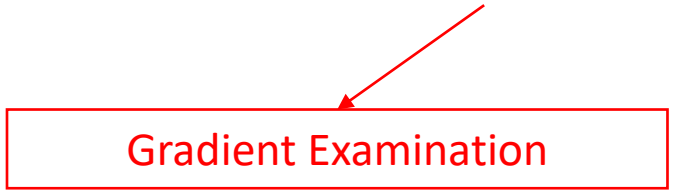
Calculate the gradient analytically



Gradient Examination

- In summary:
  - Numerical gradient: approximate, time-consuming, easy to implement!
  - Analytical gradient: accurate, fast, possibility of error in implementation!

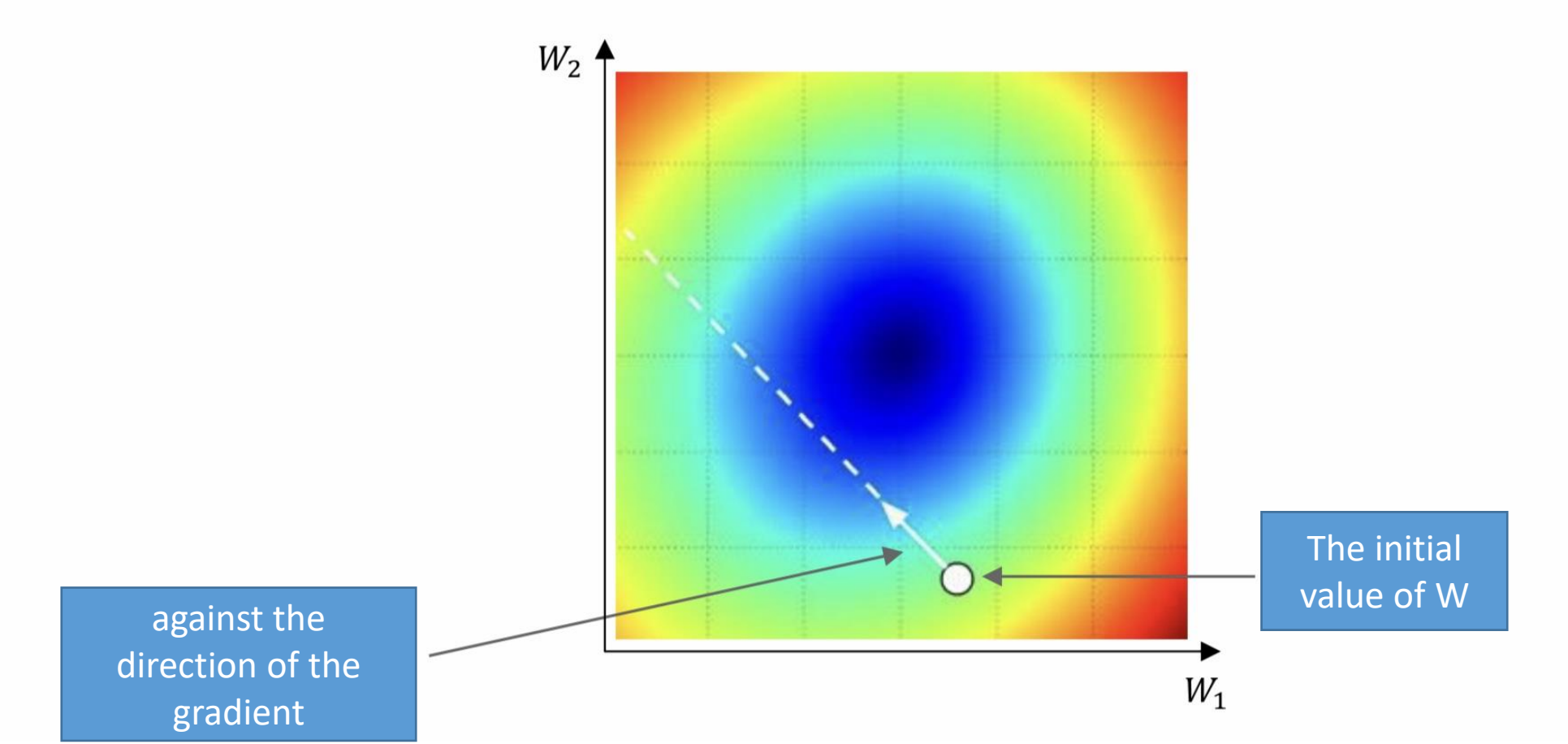
- in practice:
  - We always use the analytical gradient.
  - But to ensure the implementation's accuracy, we compare the analytical gradient with the numerical gradient.



Gradient Descent Algorithm

```
# Vanilla Gradient Descent

while True:
    gradient = evaluate_gradient(loss_fun, data, weights)
    weights += -step_size * gradient  # weight update
```

Gradient Descent Algorithm



A more efficient version of gradient descent algorithm

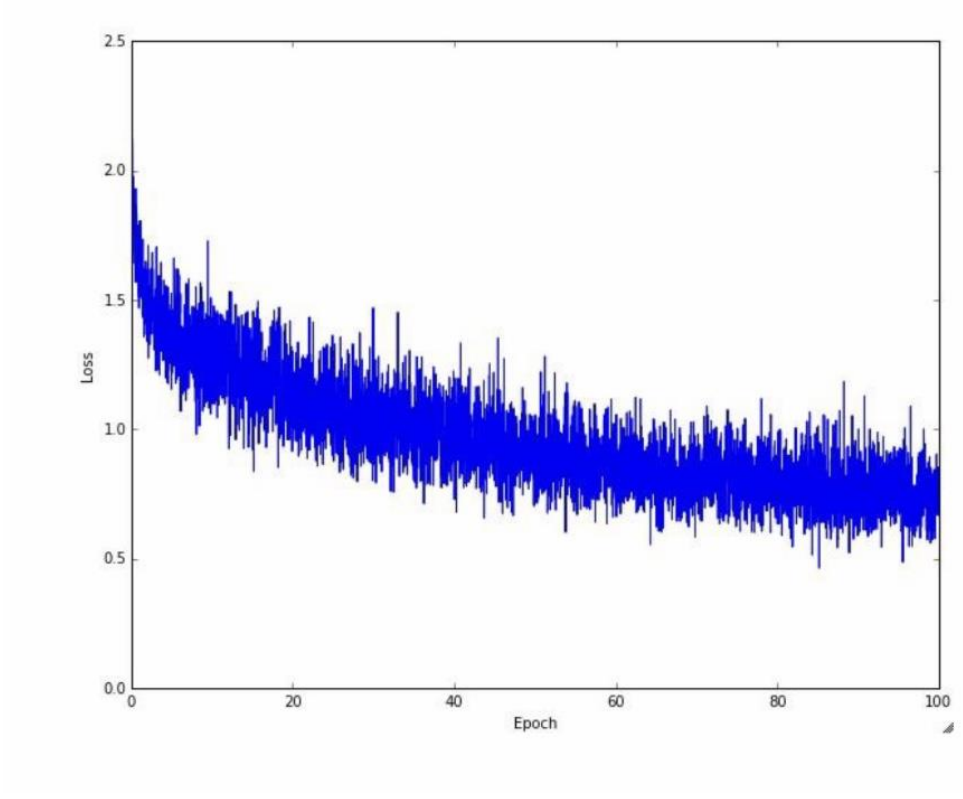
- Batch gradient descent.
  - Use only a small part of the training data to calculate the gradient of the cost function in each iteration.

```
# Mini-batch Gradient Descent

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

  - Common values for batch size: 32, 64, 128, and 256.

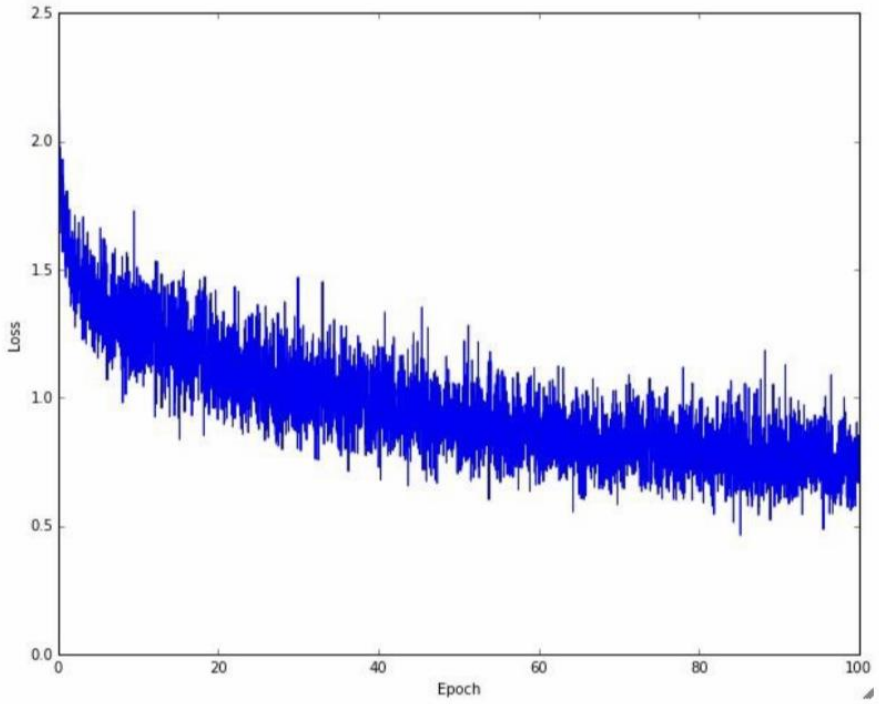
Batch Gradient Descent

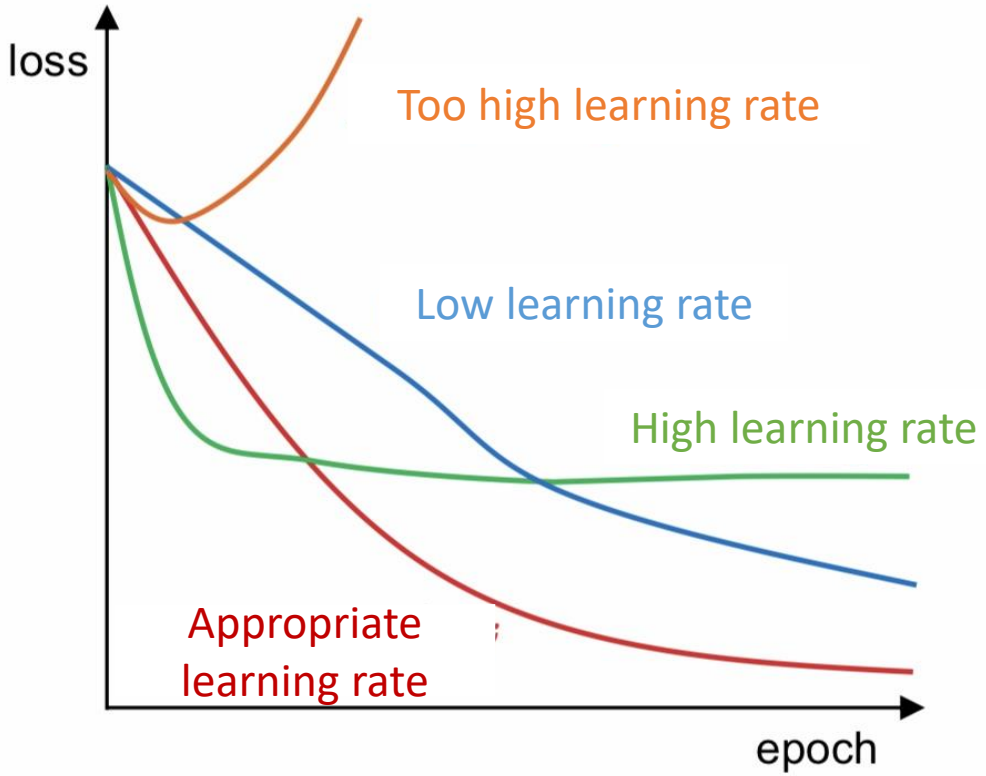
- Implementation of batch gradient descent algorithm, in order to optimize the weights of a neural network.



The cost will decrease over time.

Effect of learning rate (step size)





A more efficient version of gradient descent algorithm

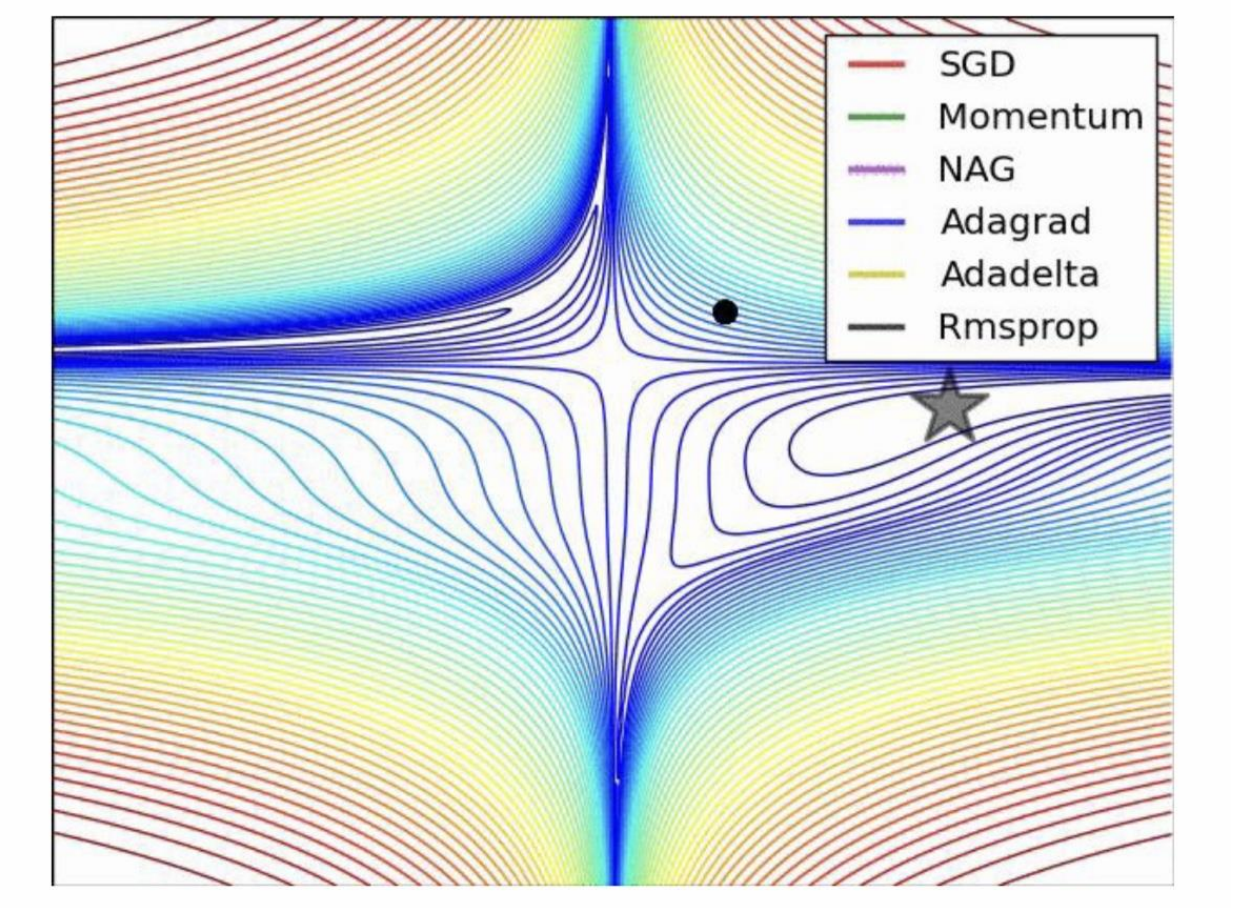
- Batch gradient descent.
  - Use only a small part of the training data to calculate the gradient of the cost function in each iteration.

```
# Mini-batch Gradient Descent

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

  - Common values for batch size: 32, 64, 128, and 256.

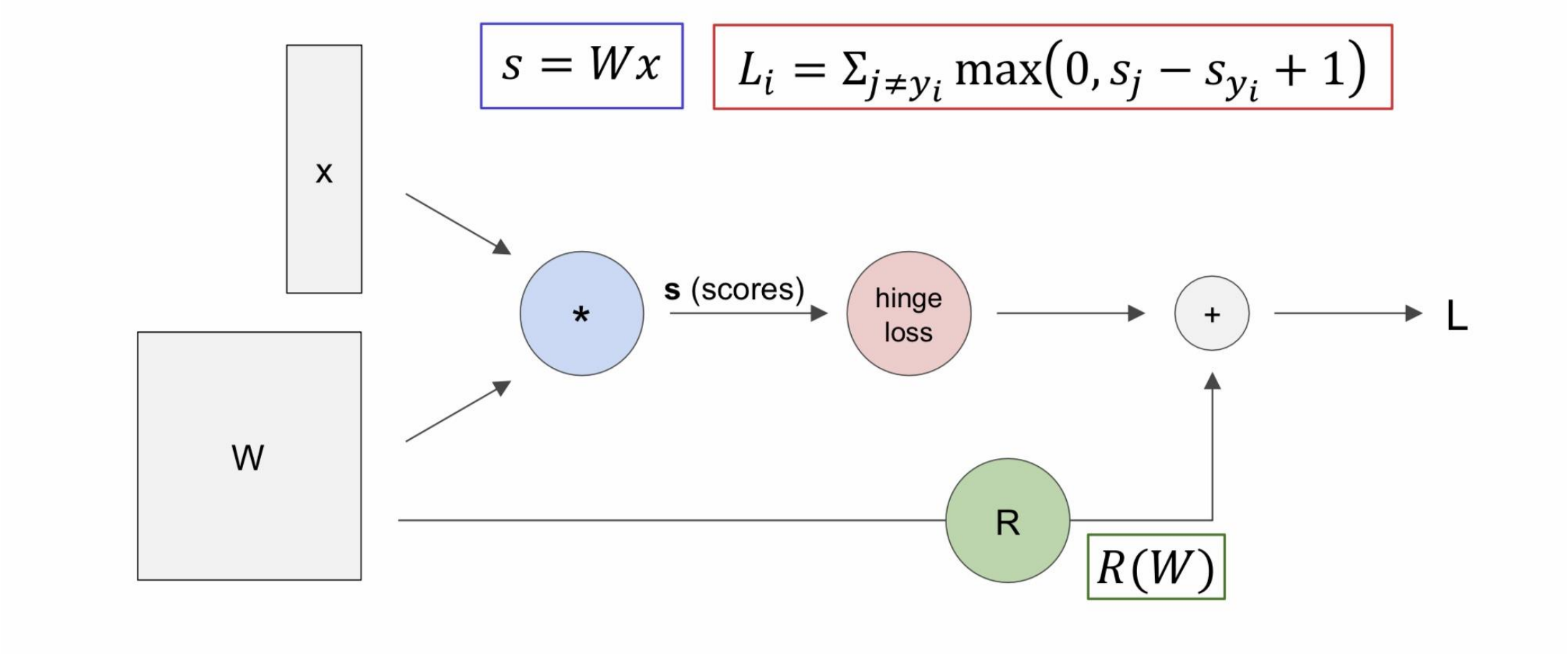
Other methods of updating the amount of weights.
momentum Adagrad Adam etc.

Update the amount of weights



Back propagation algorithm

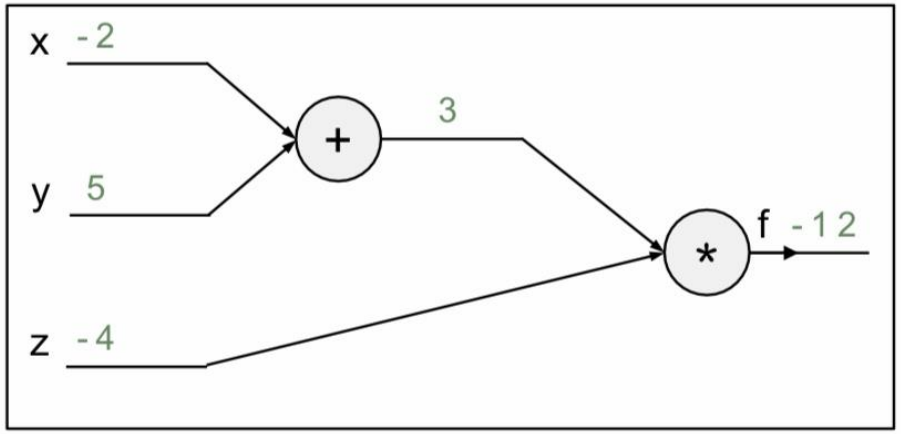
Computational Graph



Computational Graph

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

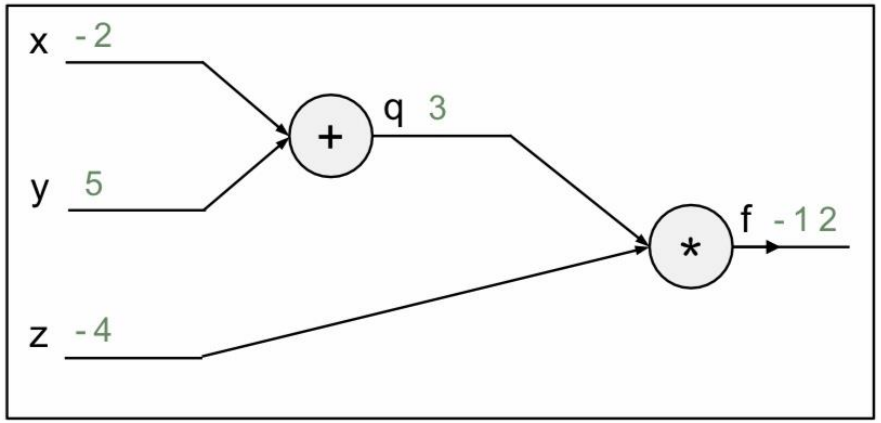
x = -2, y = 5, z = -4



Computational Graph

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

x = -2, y = 5, z = -4



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

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

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

Required values

Computational Graph

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

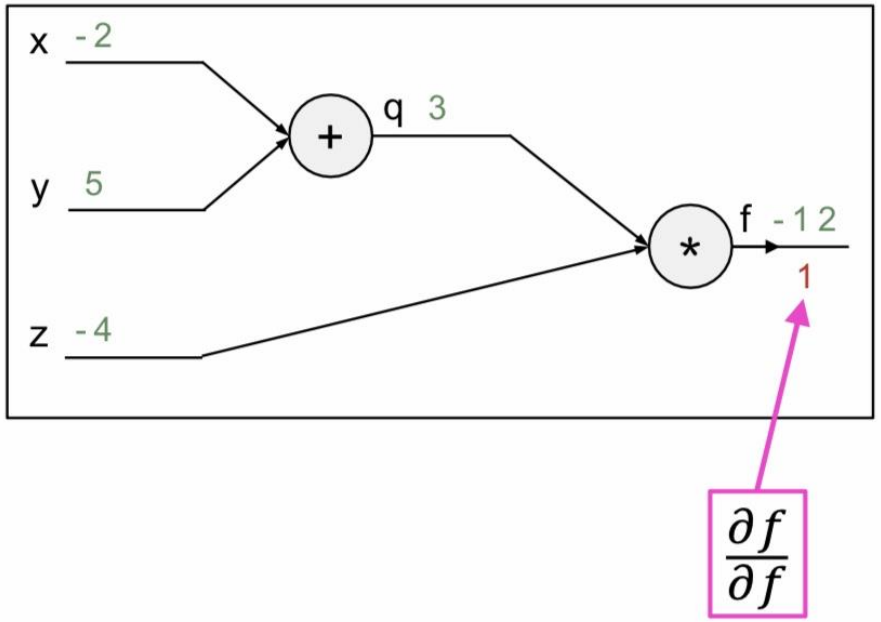
x = -2, y = 5, z = -4

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

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

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

Required values



Computational Graph

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

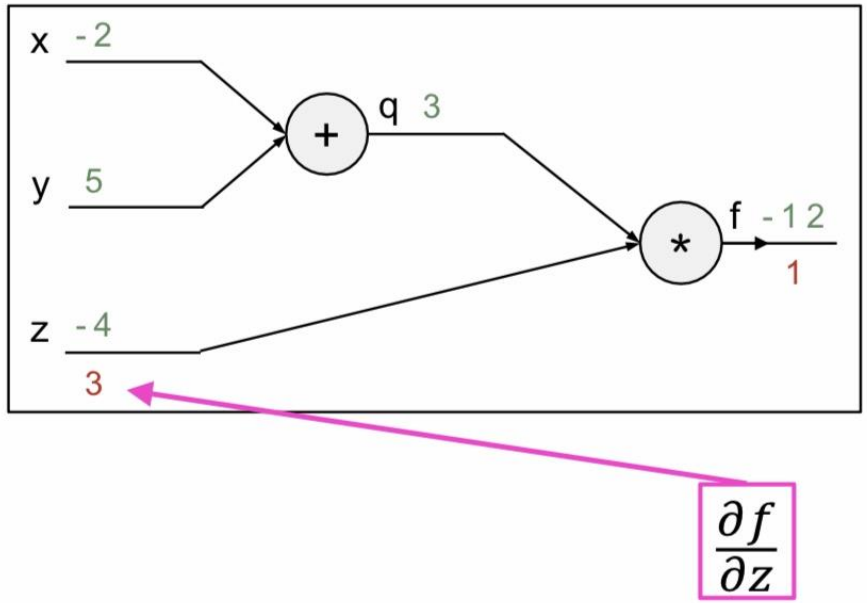
x = -2, y = 5, z = -4

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

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

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

Required values



Computational Graph

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

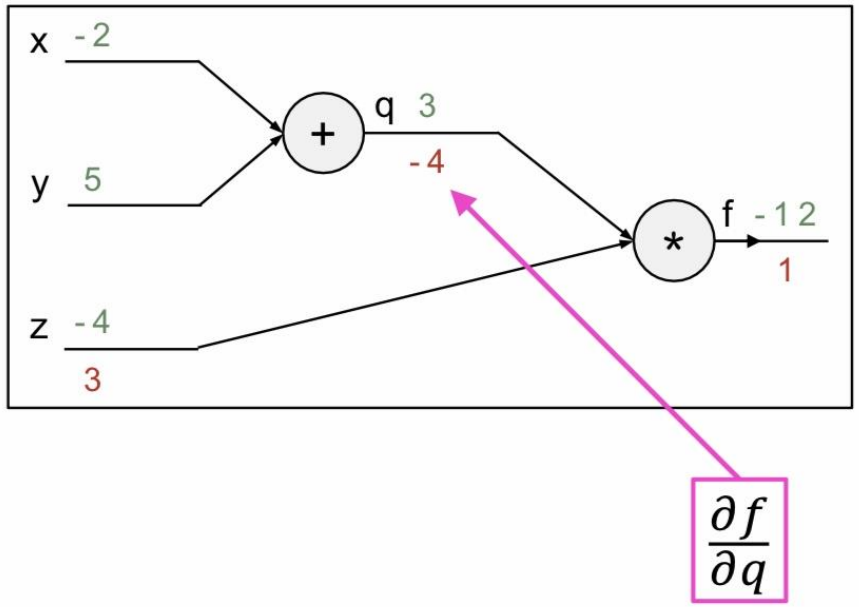
x = -2, y = 5, z = -4

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

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

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

Required values



Computational Graph

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

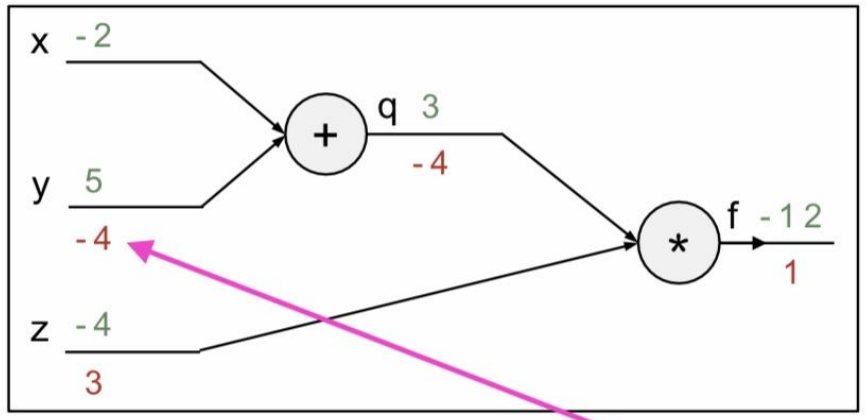
x = -2, y = 5, z = -4

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

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

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

Required values



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

Chain rule

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

Computational Graph

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

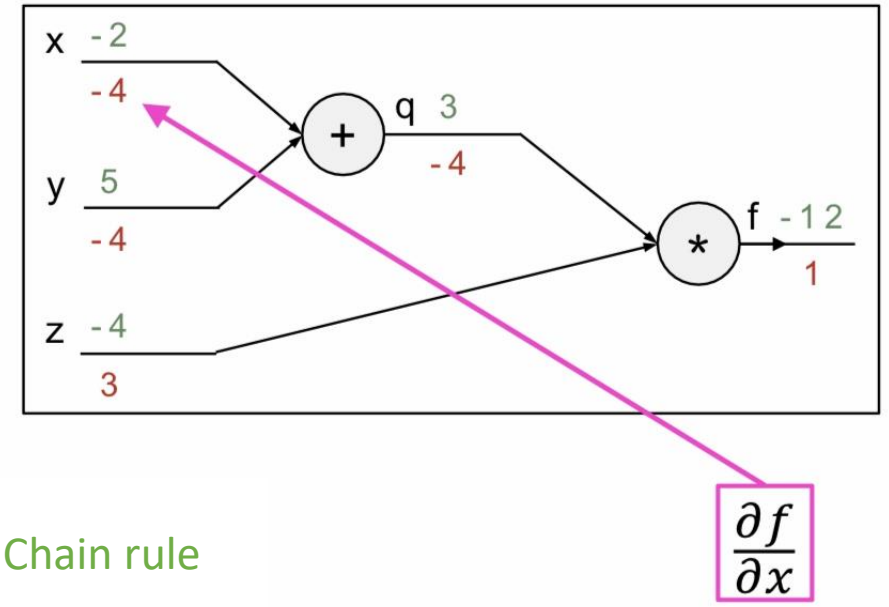
x = -2, y = 5, z = -4

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

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

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

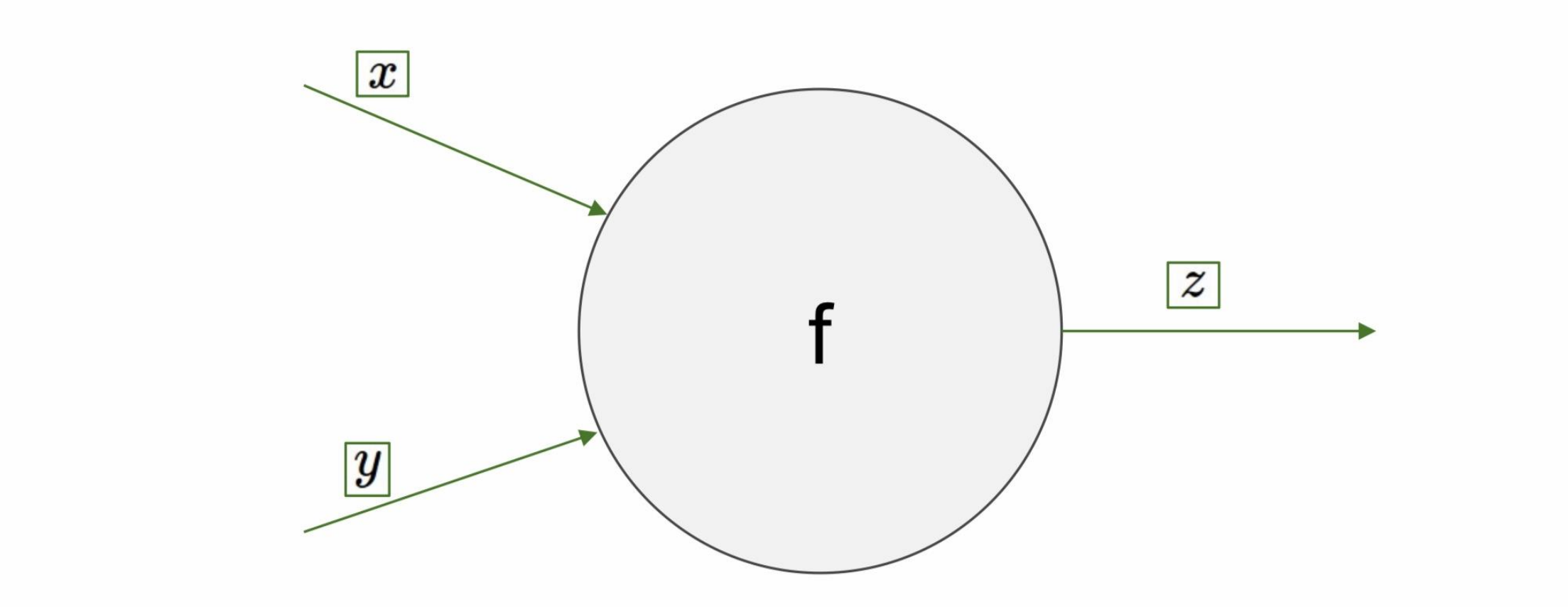
Required values



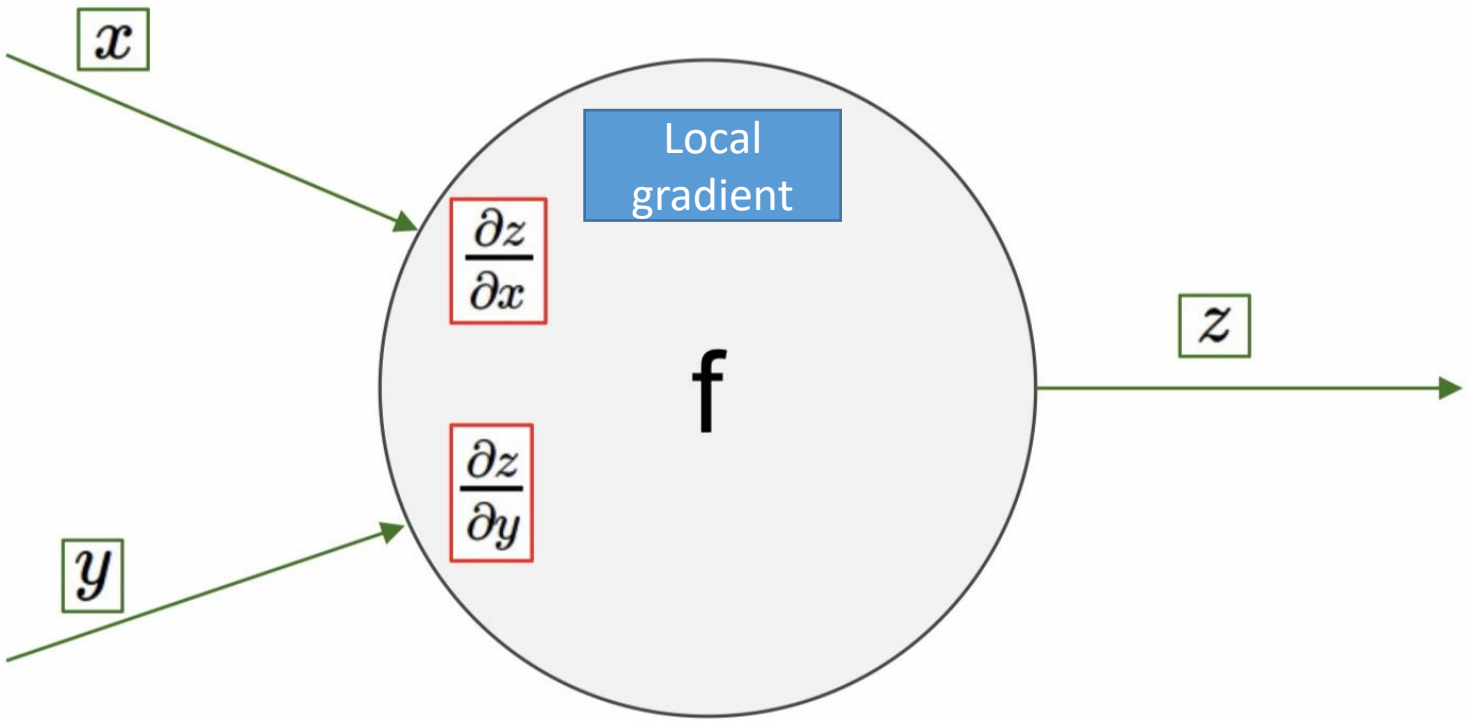
Chain rule

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

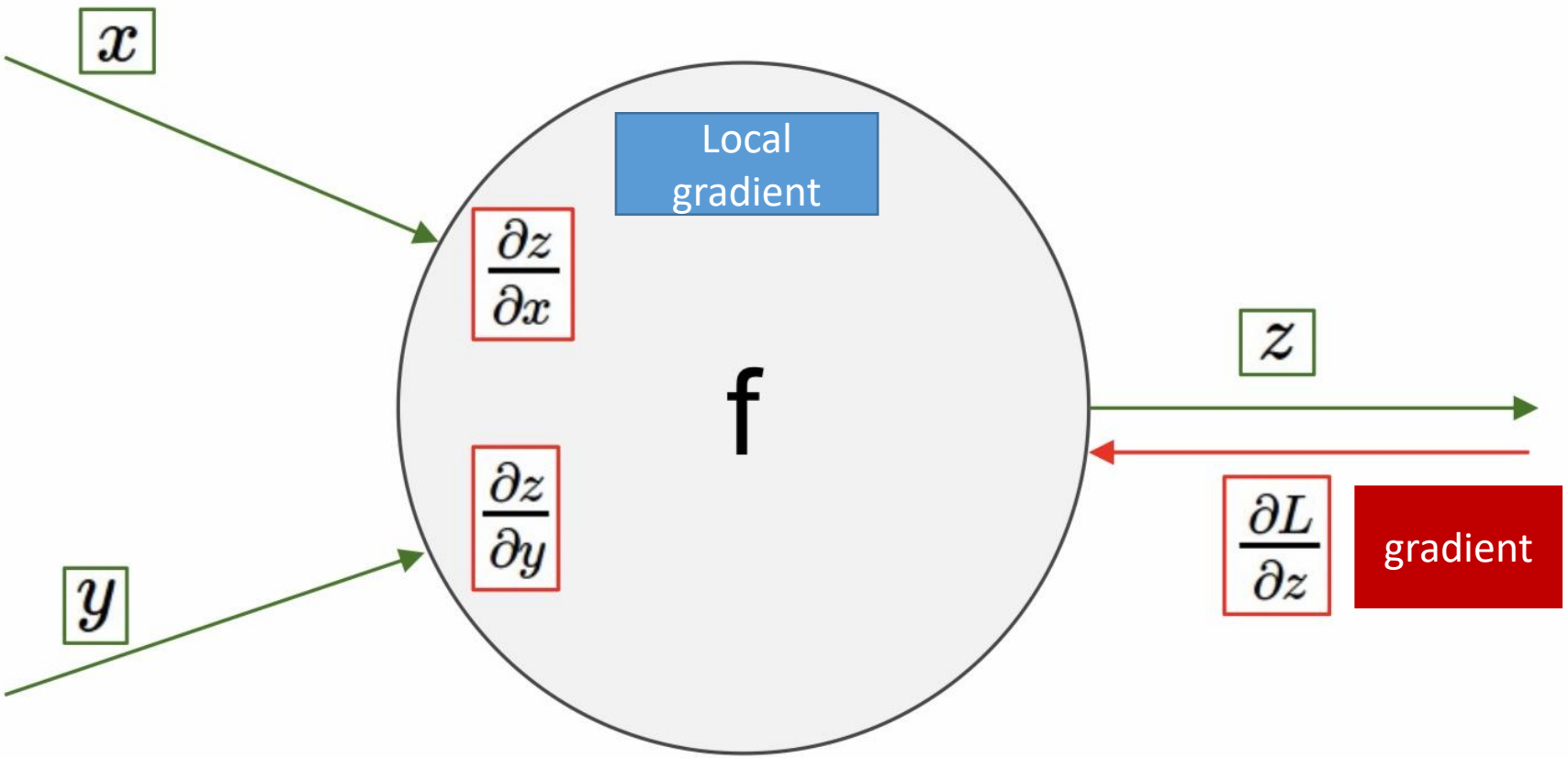
Computational Graph



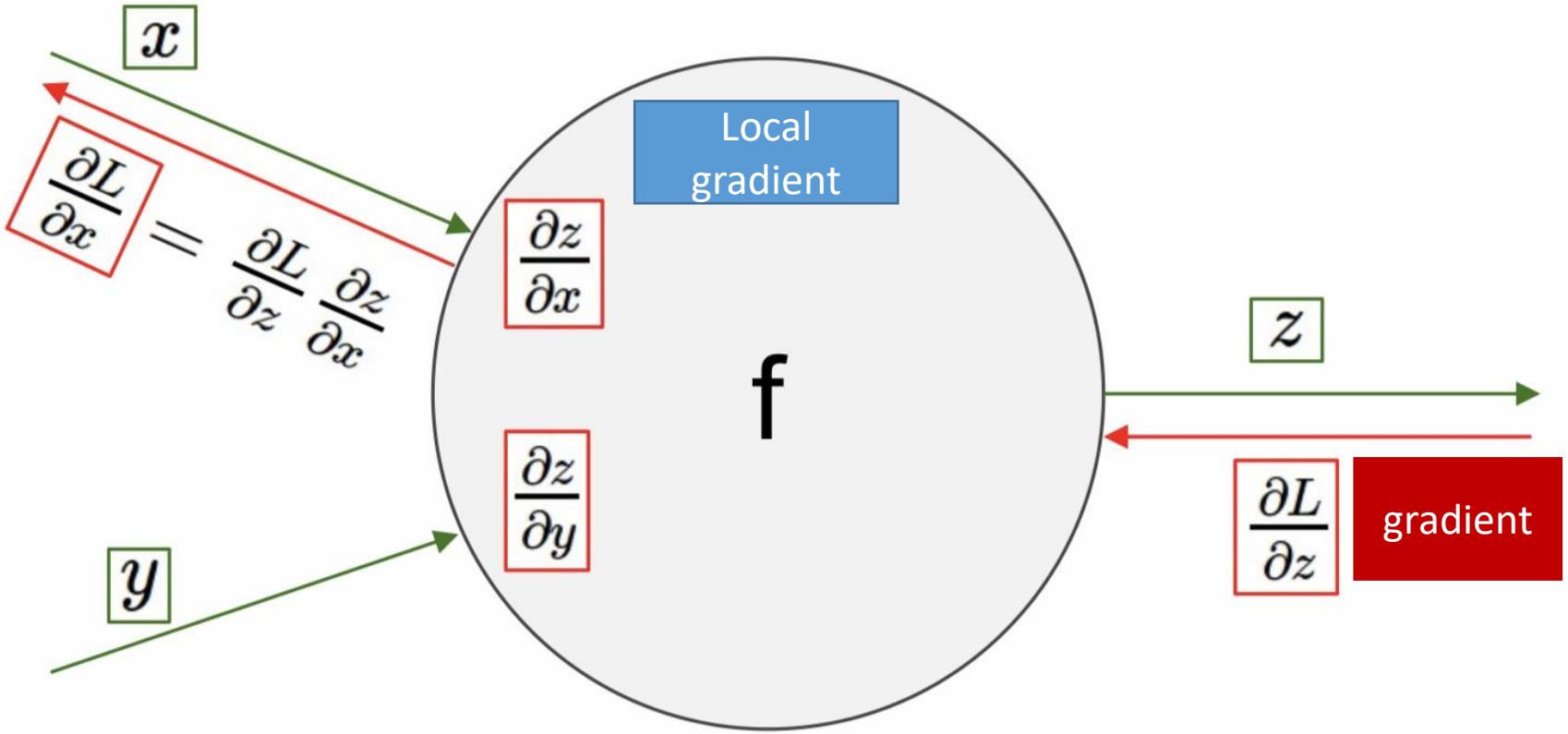
Computational Graph



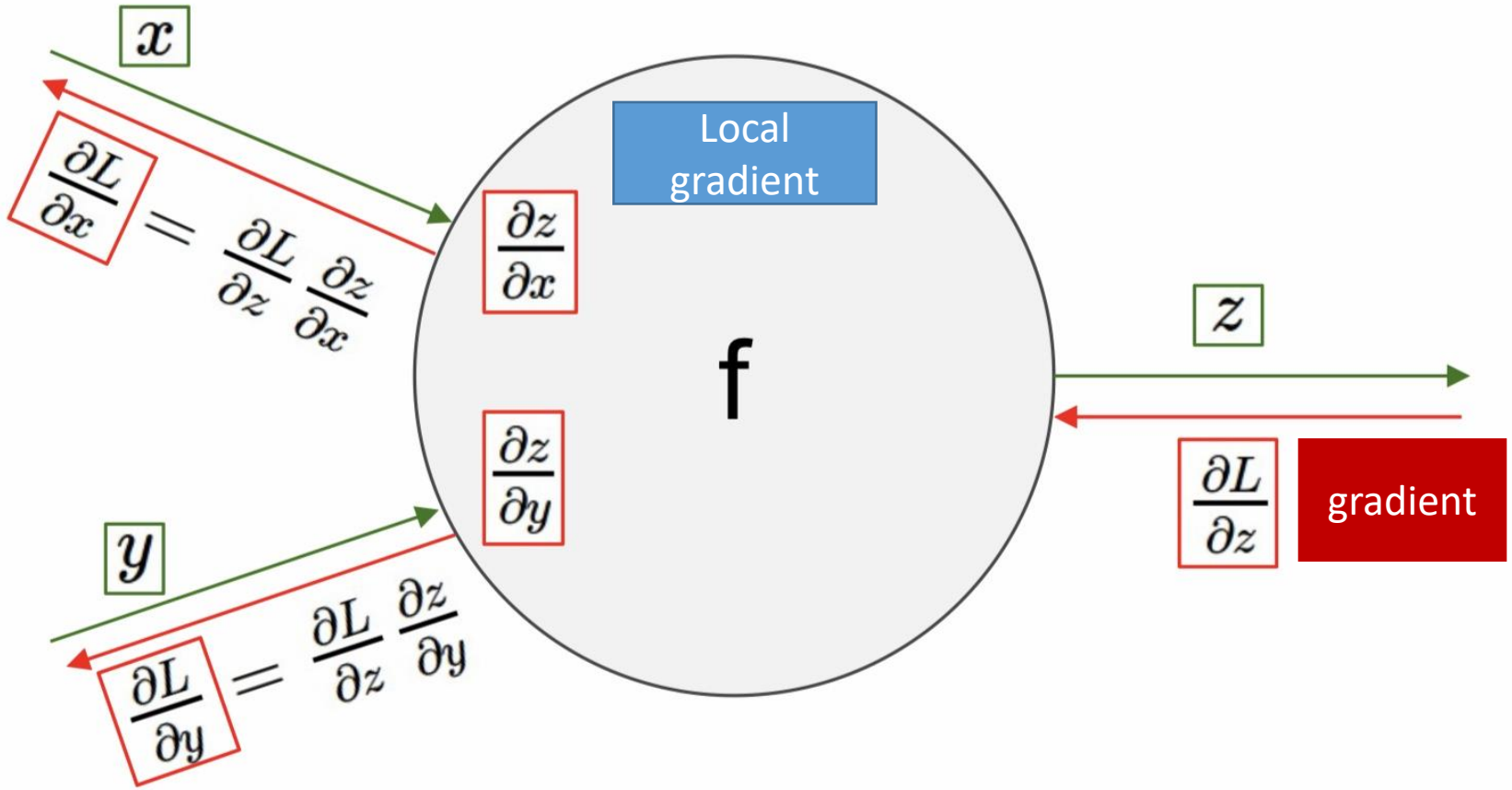
Computational Graph



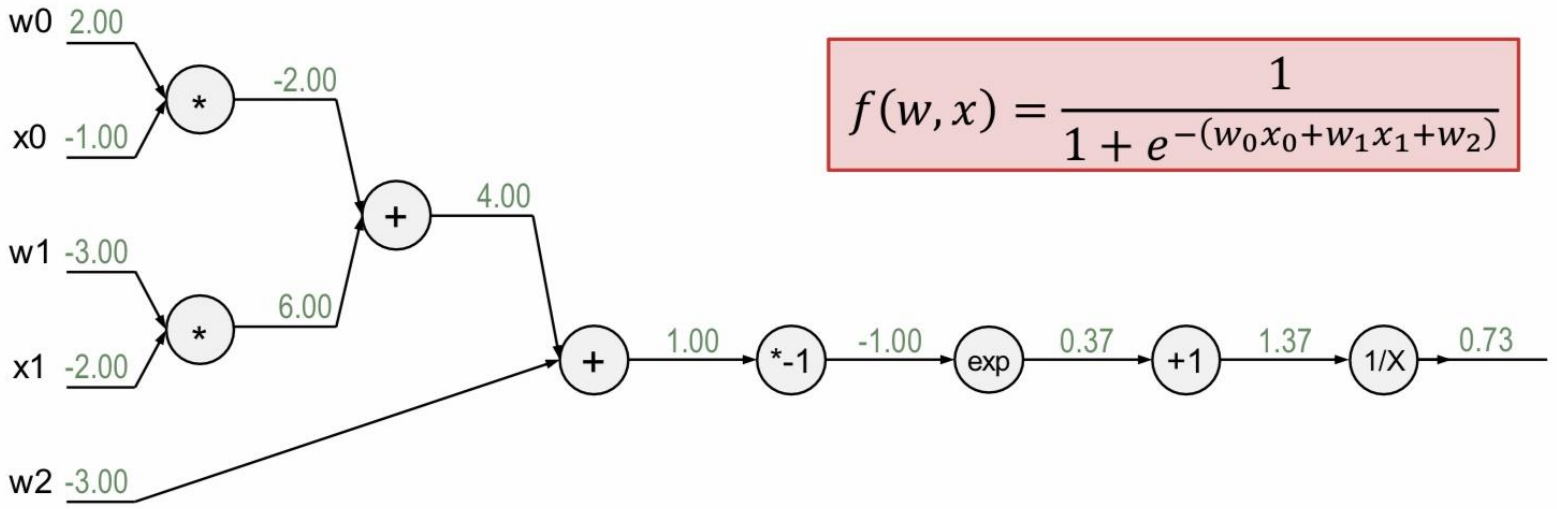
Computational Graph



Computational Graph



Computational Graph: another example



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

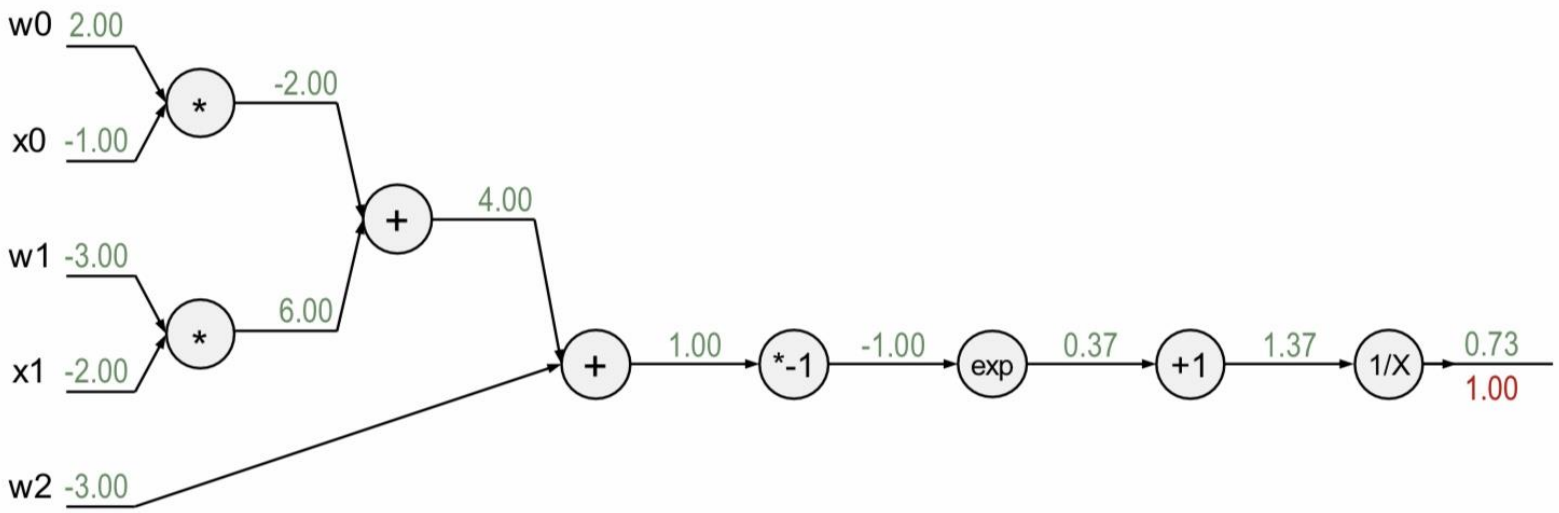
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



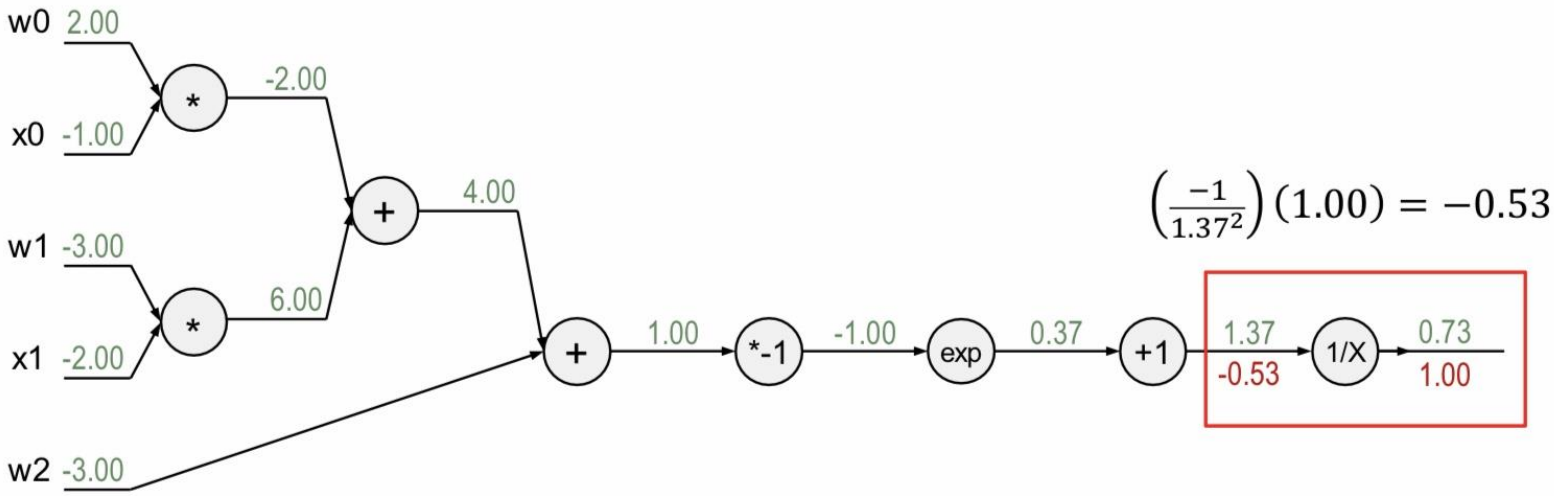
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



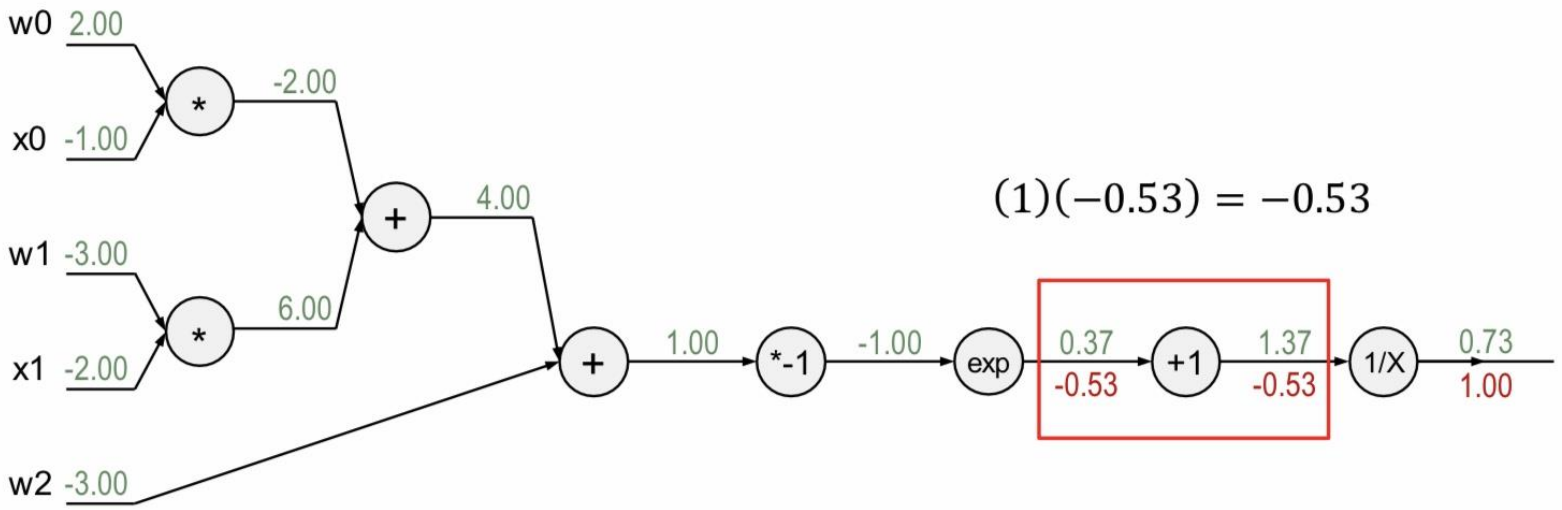
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



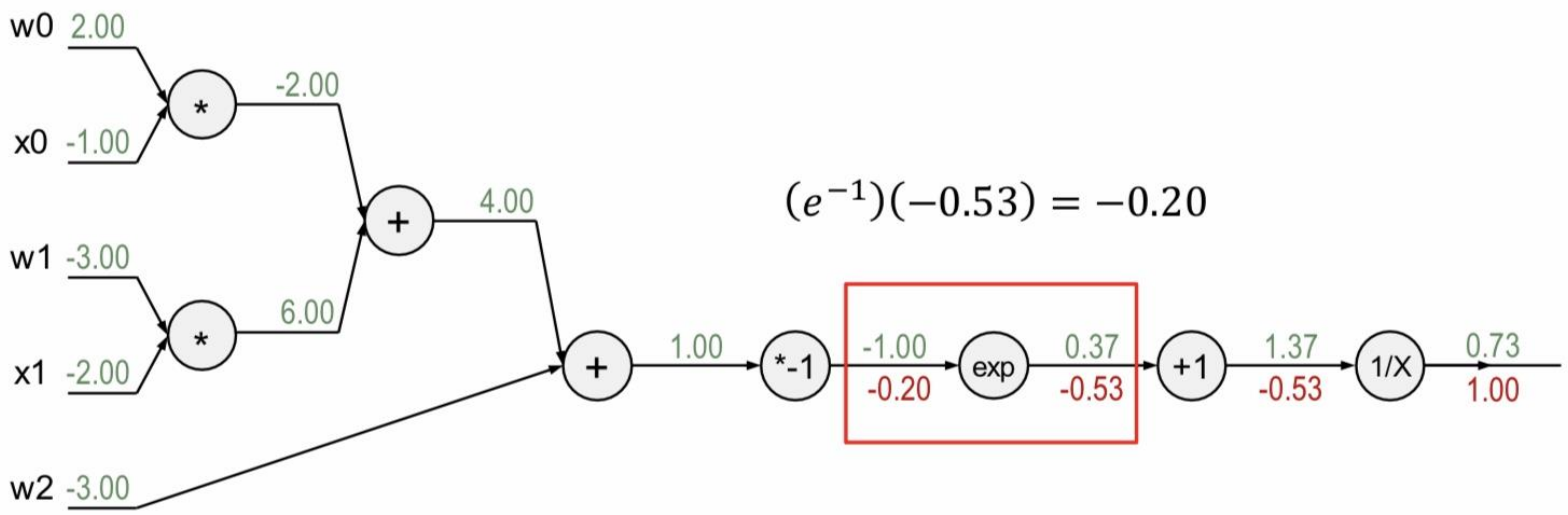
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



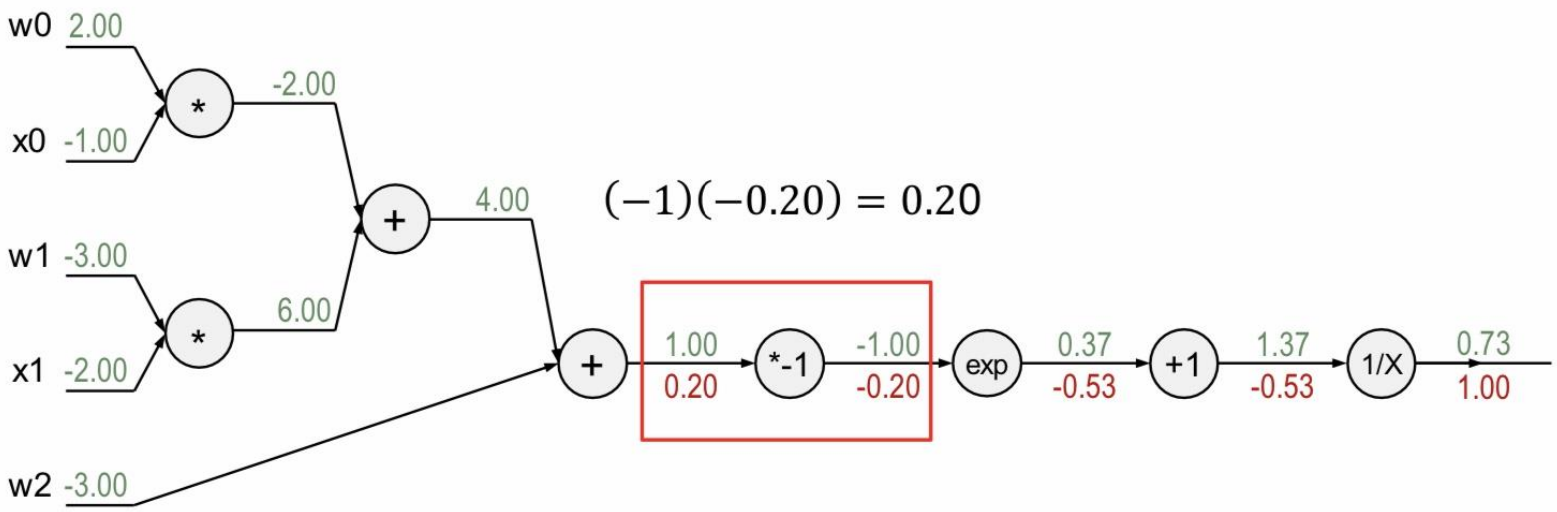
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



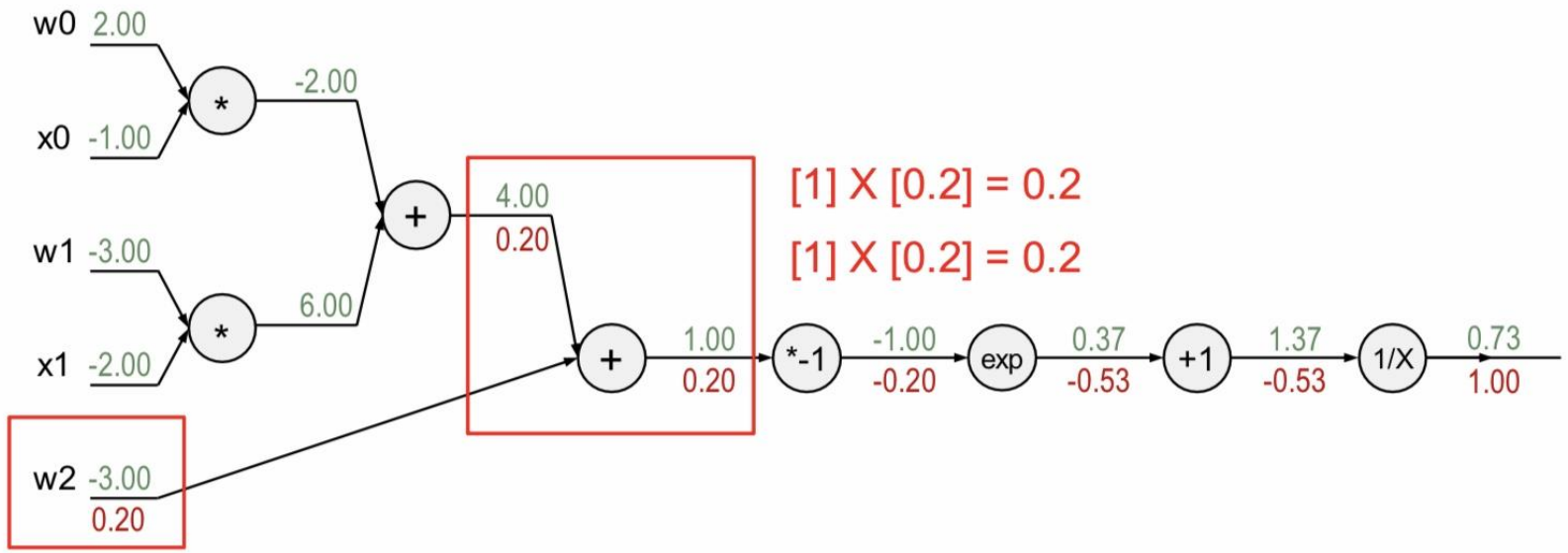
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



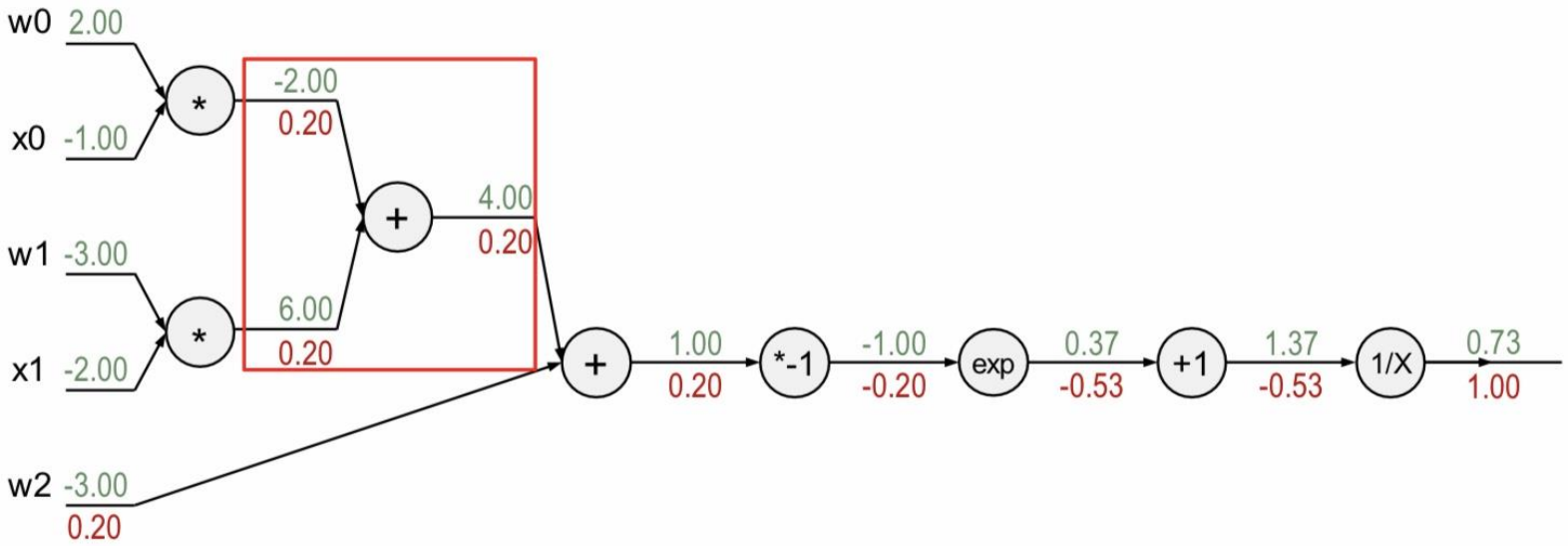
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



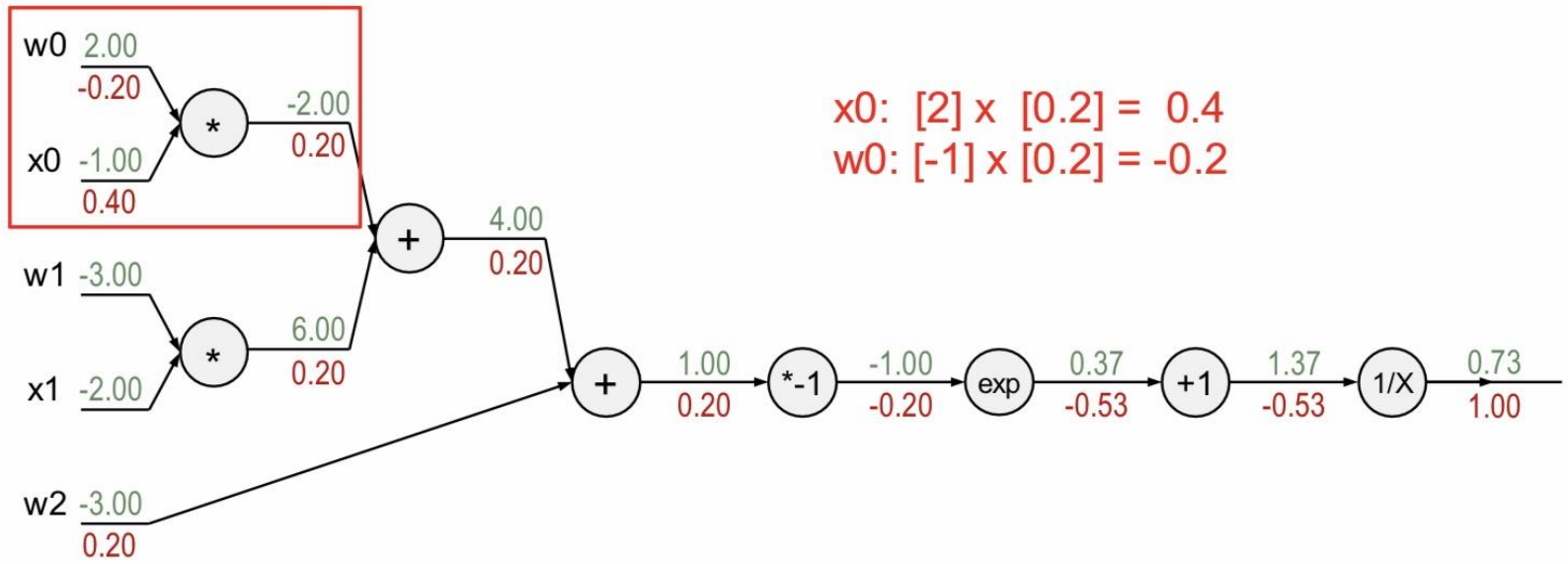
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



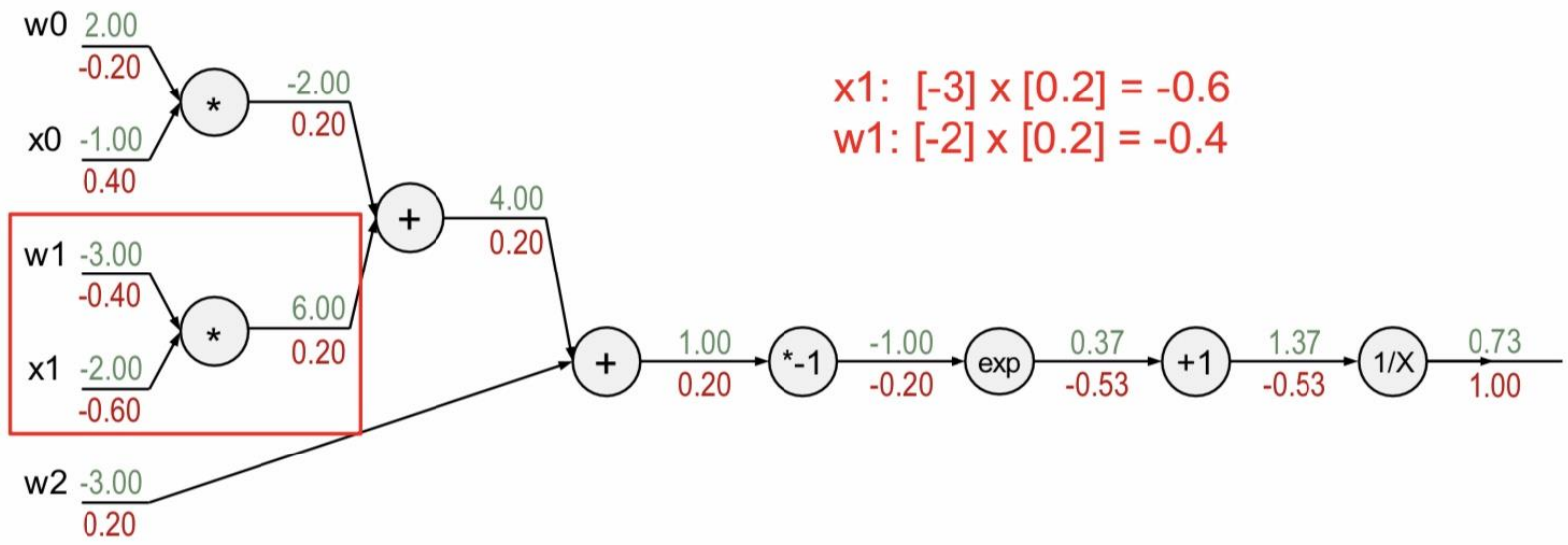
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example



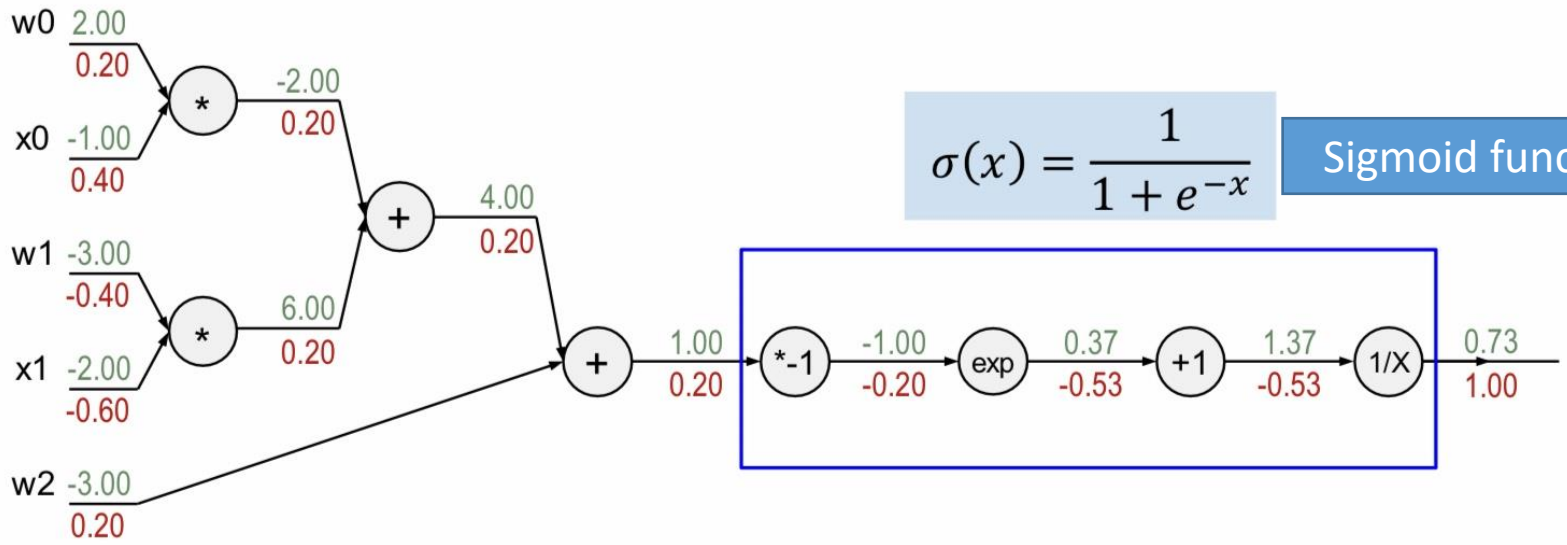
$$f(x) = e^x \quad \rightarrow \quad \frac{df}{dx} = e^x$$

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

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

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

Computational Graph: another example

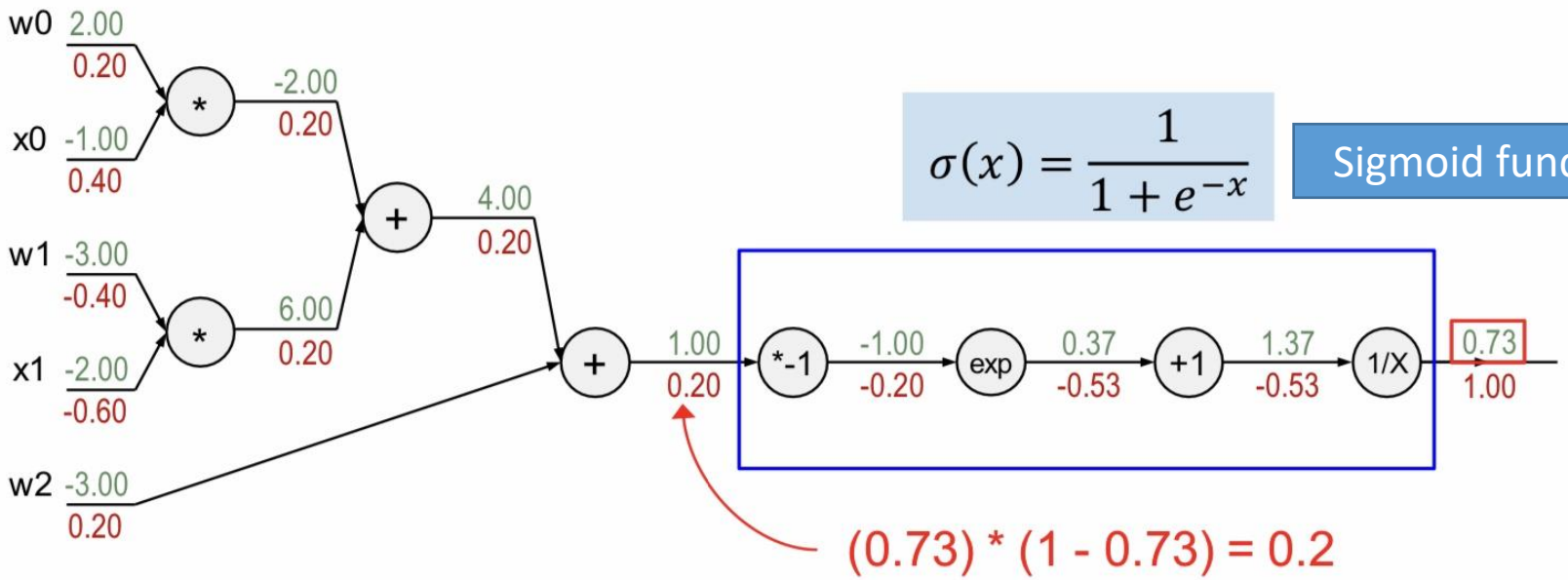


$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Sigmoid function

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

Computational Graph: another example



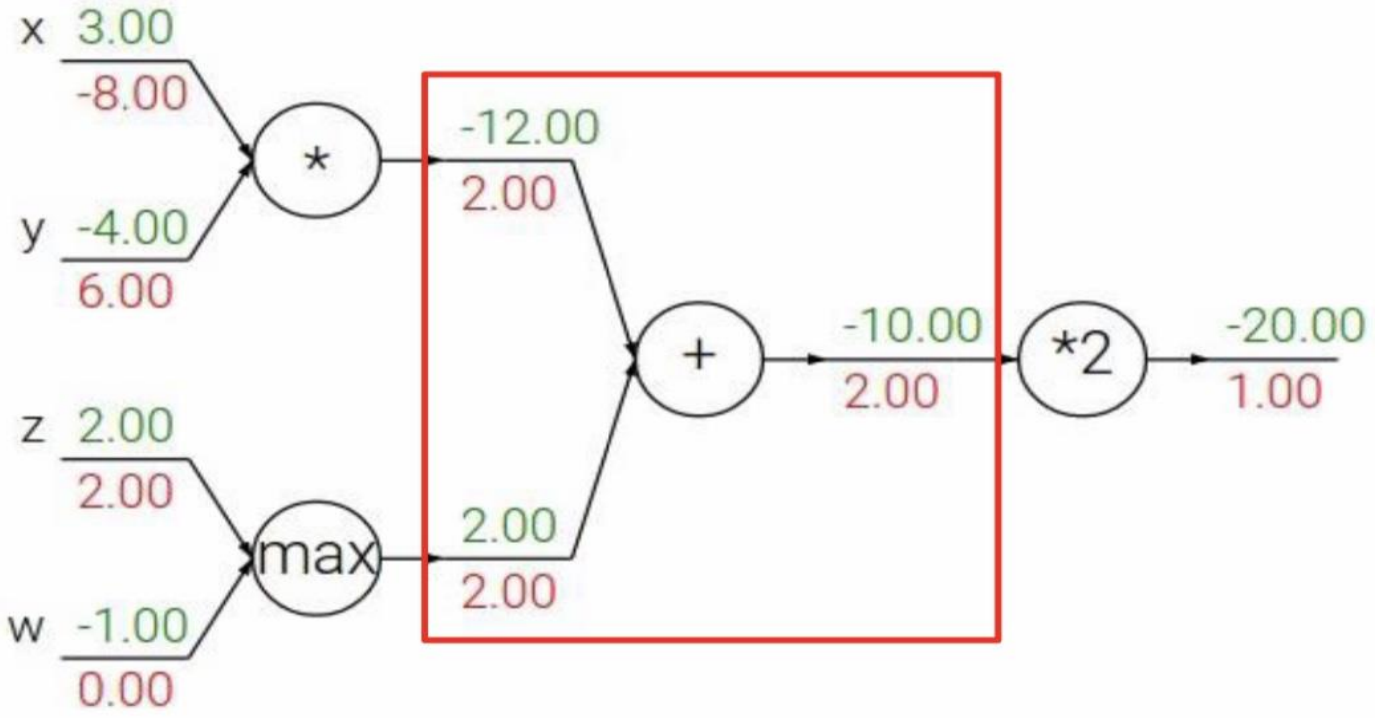
$$\sigma(x) = \frac{1}{1+e^{-x}}$$

Sigmoid function

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

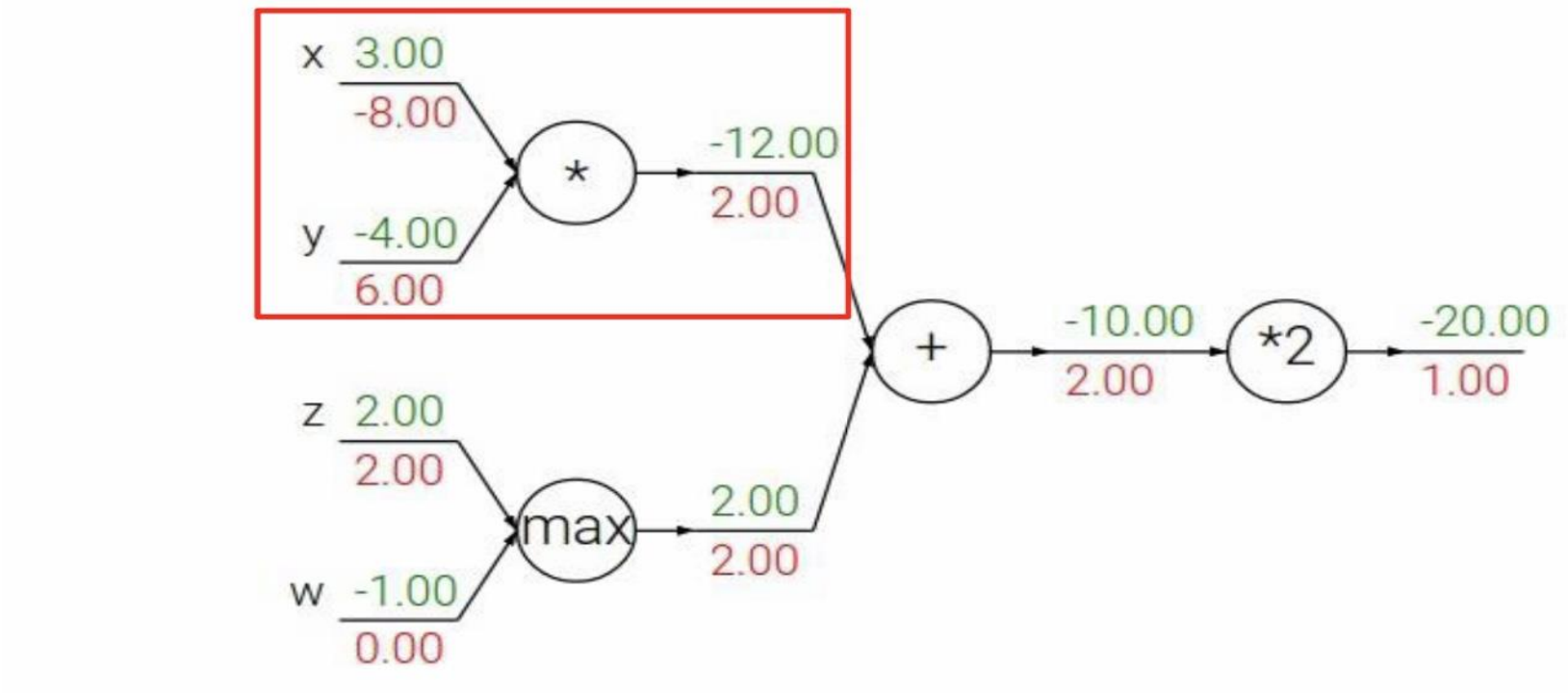
Some algorithms in backward calculations

- Mathematical sum. Gradient distributor!



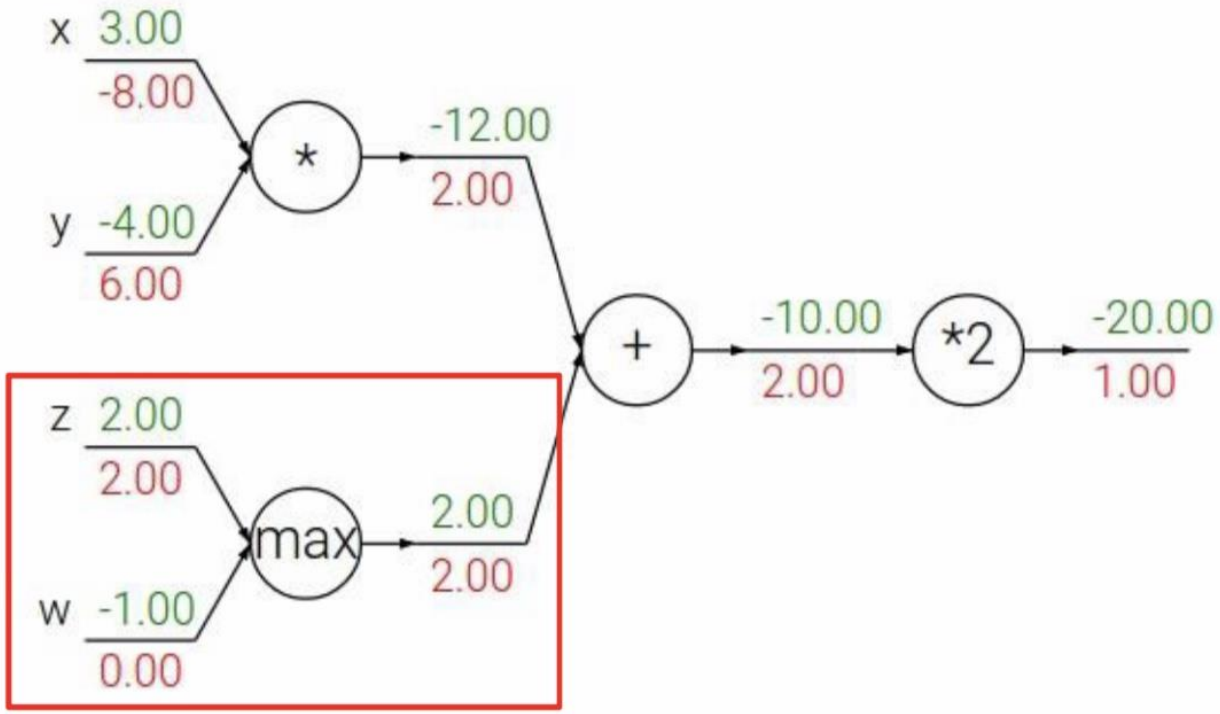
Some algorithms in backward calculations

- multiplication. Gradient shifter!



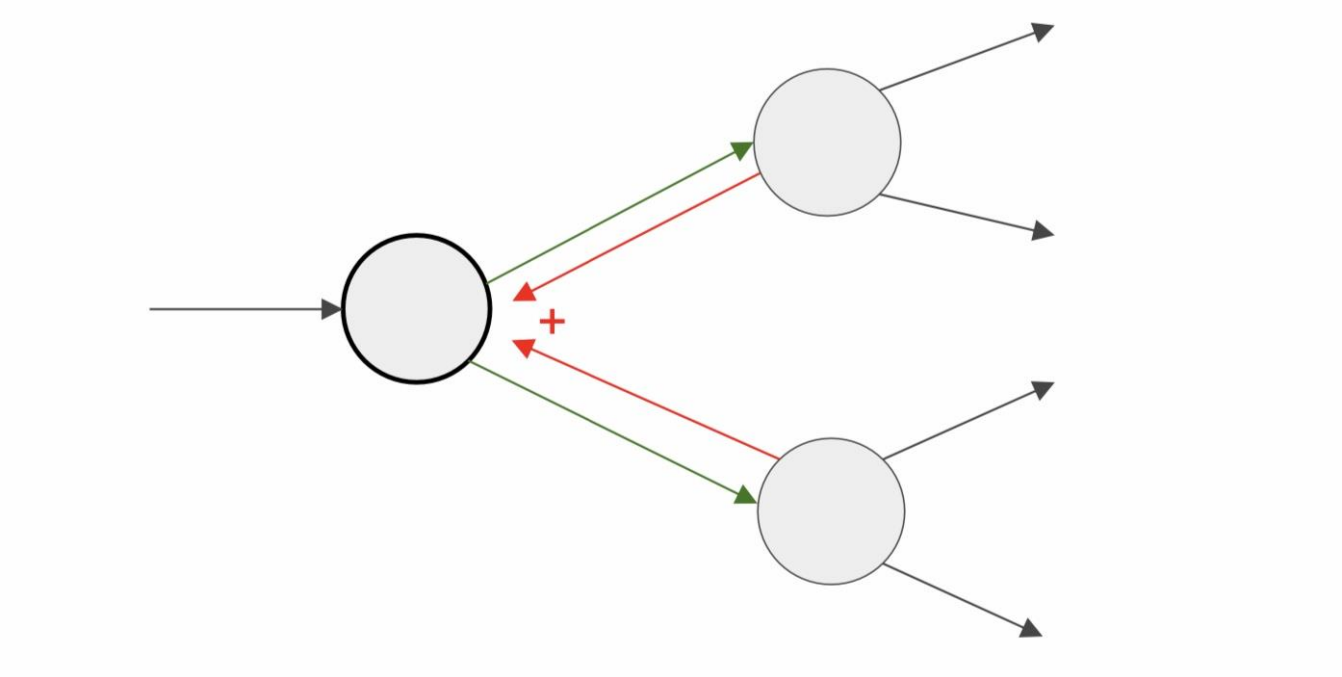
Some algorithms in backward calculations

- Maximization operation. Gradient Routing!

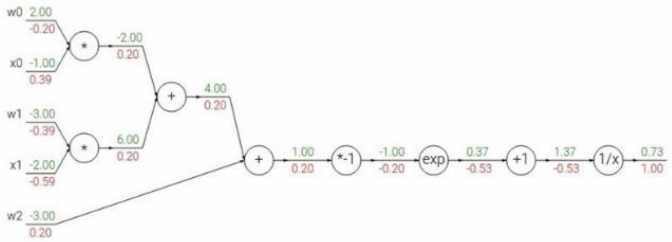


Some algorithms in backward calculations

- Aggregation of gradients in bifurcations.

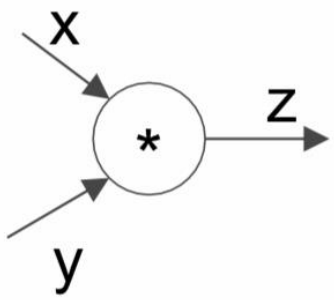


Implementation: forward and backward calculations.




```
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```

Implementation: forward and backward calculations.

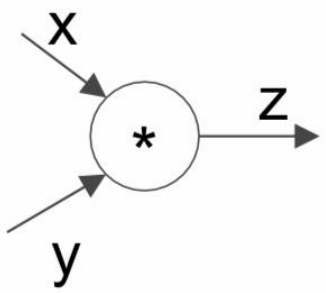



```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        return z
    def backward(dz):
        # dx = ... #todo
        # dy = ... #todo
        return [dx, dy]
```

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

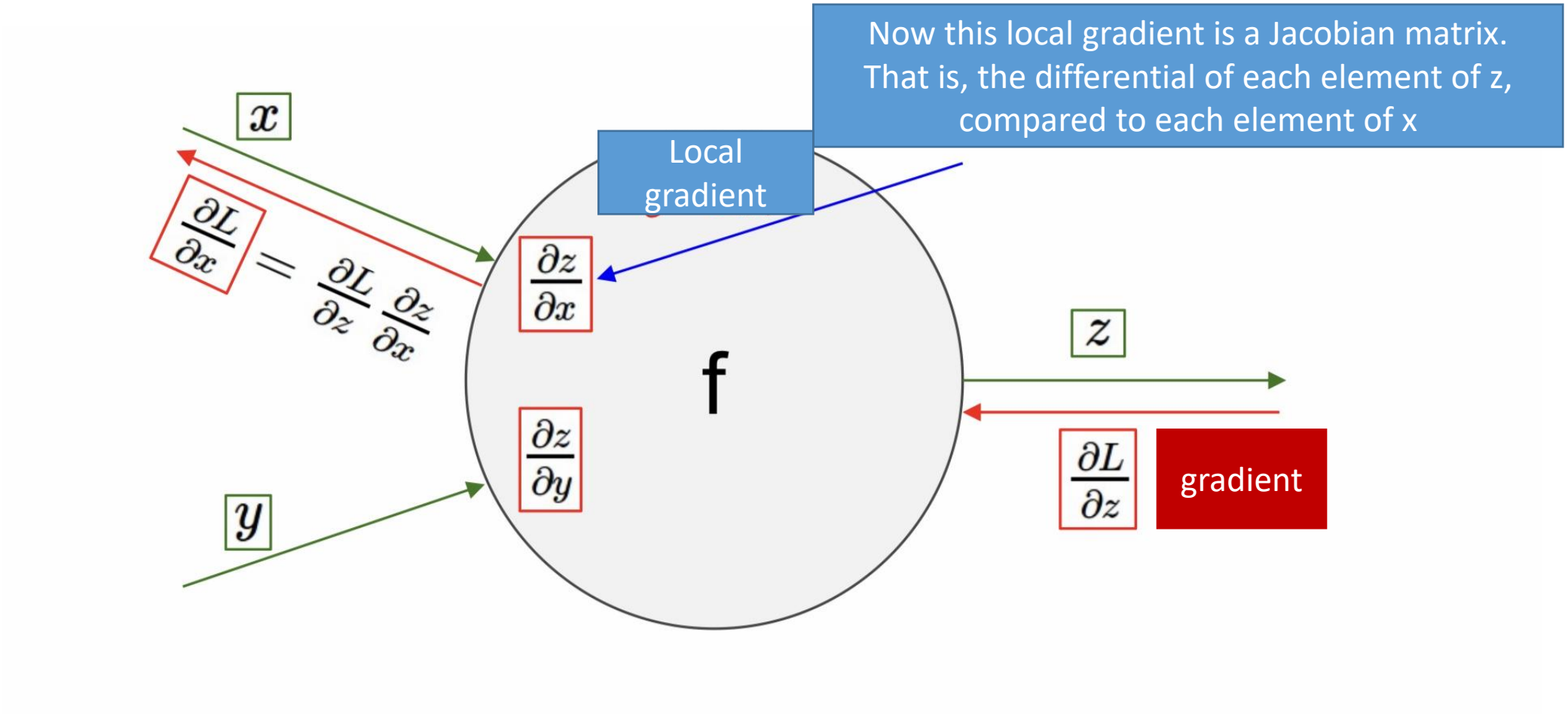
$\frac{\partial L}{\partial x}$

Implementation: forward and backward calculations.



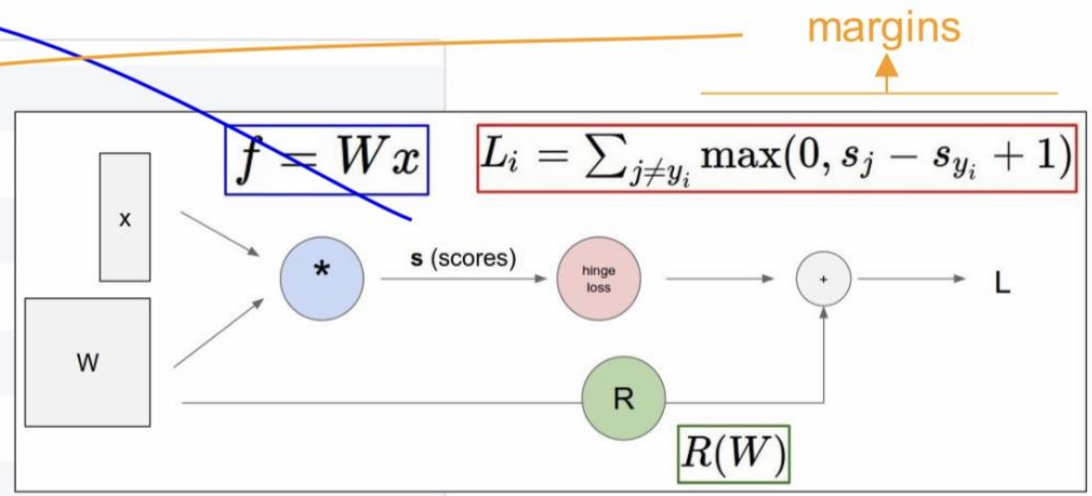
```
class MultiplyGate(object):
    def forward(x,y):
        z = x*y
        self.x = x # must keep these around!
        self.y = y
        return z
    def backward(dz):
        dx = self.y * dz # [dz/dx * dL/dz]
        dy = self.x * dz # [dz/dy * dL/dz]
        return [dx, dy]
```

Gradients for Vectors



Implementation of Cost Function

```
# receive W (weights), X (data)
# forward pass (we have 8 lines)
scores = #...
margins = #...
data_loss = #...
reg_loss = #...
loss = data_loss + reg_loss
# backward pass (we have 5 lines)
dmargins = # ... (optionally, we go direct to dscores)
dscores = #...
dW = #...
```



Conclusion

- The number of parameters in a neural network can be very large:
  - It is impossible to write the relationship related to the gradient of all parameters manually!
- Backpropagation: Applying the chain rule recursively, along a computational graph, to compute the gradient of the cost function with respect to parameters, inputs, and intermediate values.
- Computational graph: A graph structure where each node implements forward and backward calculations.
  - Forward calculations: calculate the result of an operation and store the intermediate values needed to calculate the gradient.
  - Backward calculations: using the chain rule to calculate the gradient of the cost function with respect to the inputs.

Neural Networks

Neural Networks

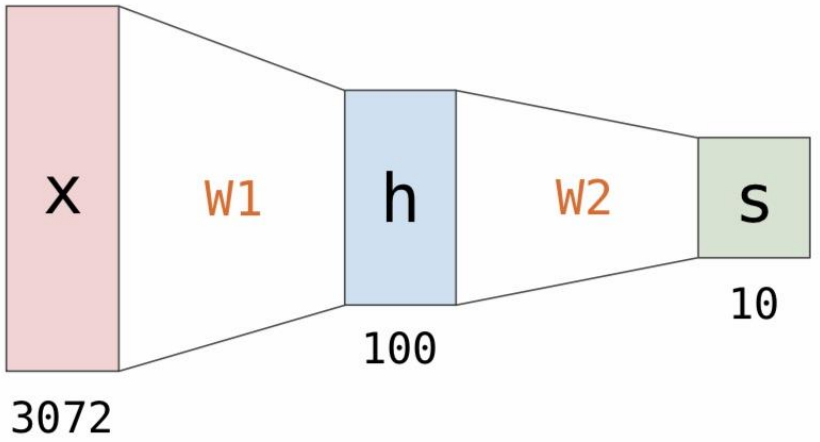
Neural Networks

$$f = Wx$$

(previously) linear score function

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

(now) 2-layer neural network



Neural Networks

$$f = Wx$$

(previously) linear score function

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

(now) 2-layer neural network

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

Or 3-layer neural network

Neural Networks

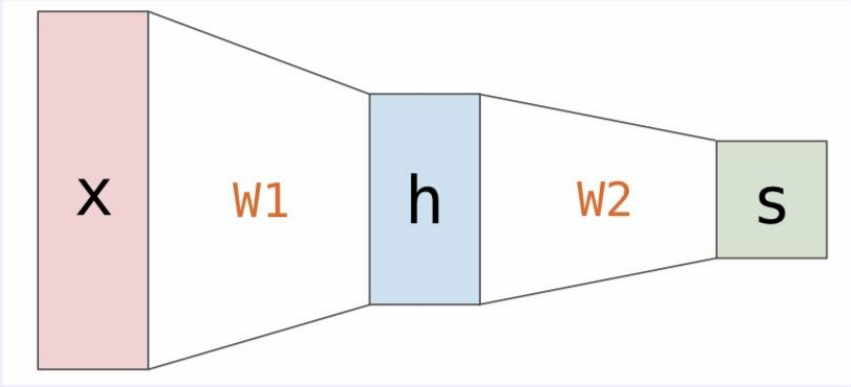
- Implementation of the training of a 2-layer neural network. (in 11 lines)

```
01.  X = np.array([ [0,0,1],[0,1,1],[1,0,1],[1,1,1] ])
02.  y = np.array([[0,1,1,0]]).T
03.  syn0 = 2*np.random.random((3,4)) - 1
04.  syn1 = 2*np.random.random((4,1)) - 1
05.  for j in xrange(60000):
06.      l1 = 1/(1+np.exp(-(np.dot(X,syn0))))
07.      l2 = 1/(1+np.exp(-(np.dot(l1,syn1))))
08.      l2_delta = (y - l2)*(l2*(1-l2))
09.      l1_delta = l2_delta.dot(syn1.T) * (l1 * (1-l1))
10.      syn1 += l1.T.dot(l2_delta)
11.      syn0 += X.T.dot(l1_delta)
```

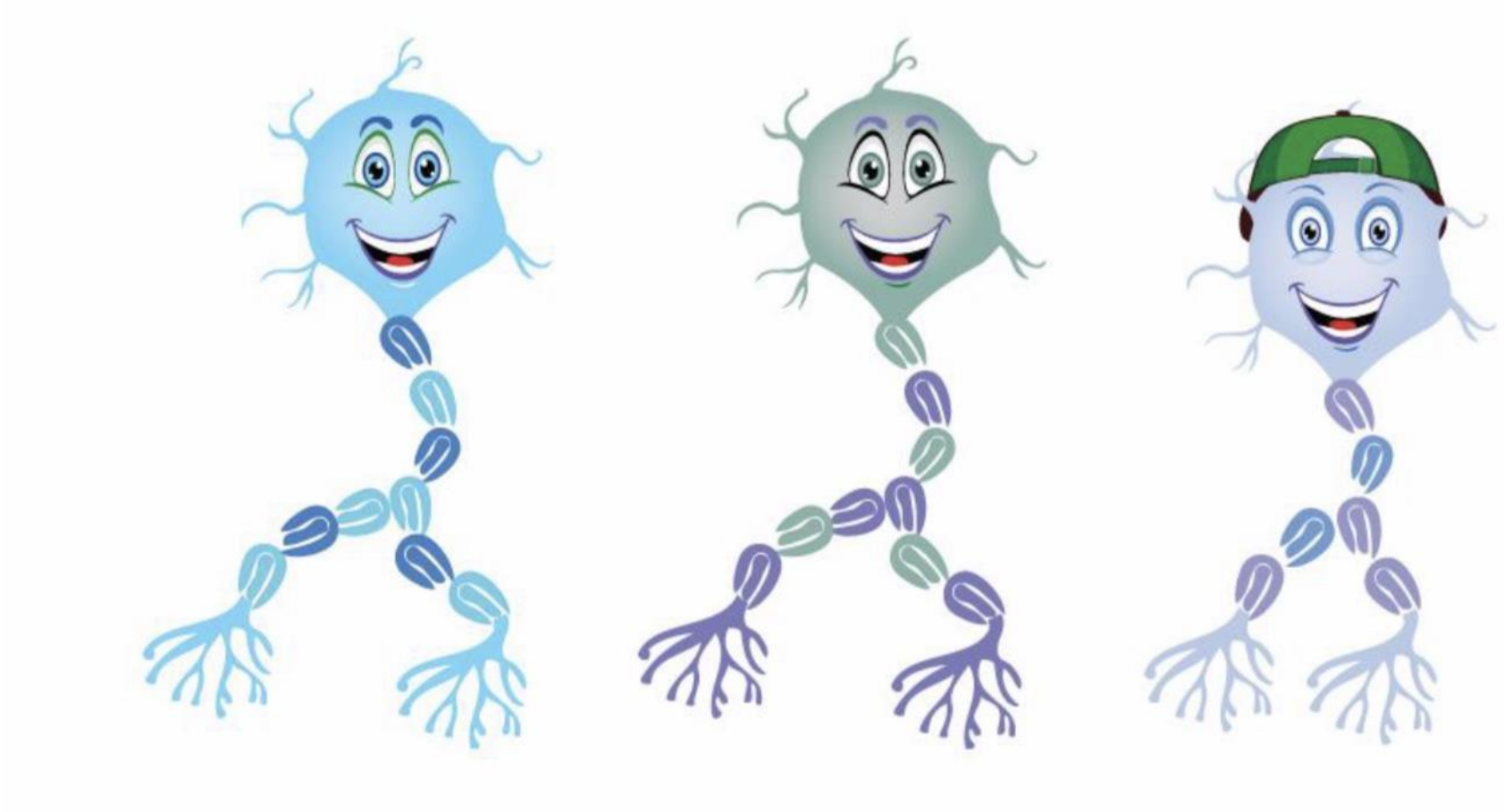
from @iamtrask, http://iamtrask.github.io/2015/07/12/basic-python-network/

Implementation of a 2-layer neural network

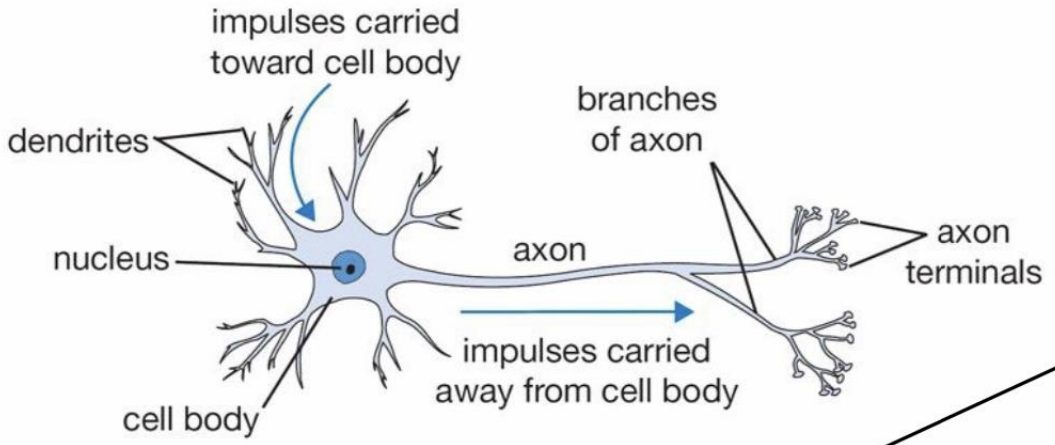
```
# receive W1, W2, b1, b2 (weights/biases), X (data)

# forward pass:
h =          #... function of X, W1, b1
scores =     #... function of h, W2, b2
loss =       #... (several lines of code to evaluate Softmax loss)

# backward pass:
dscores =           #...
dh, dW2, db2 =      #...
dW1, db1 =          #...
```

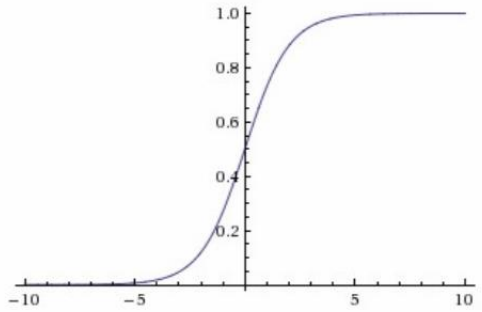


Neurons and Neural Networks



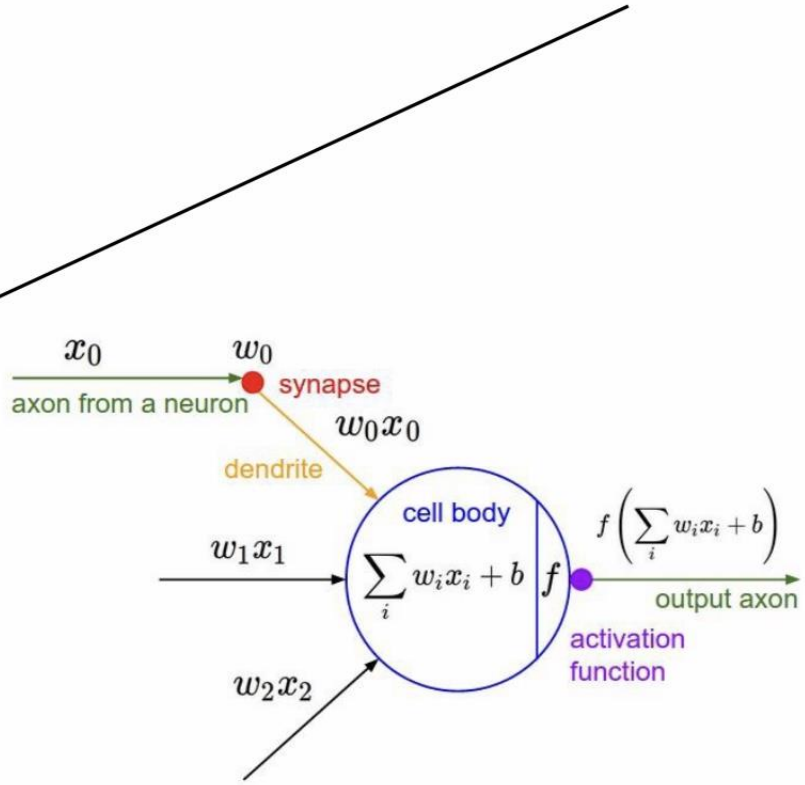
Neurons and Neural Networks



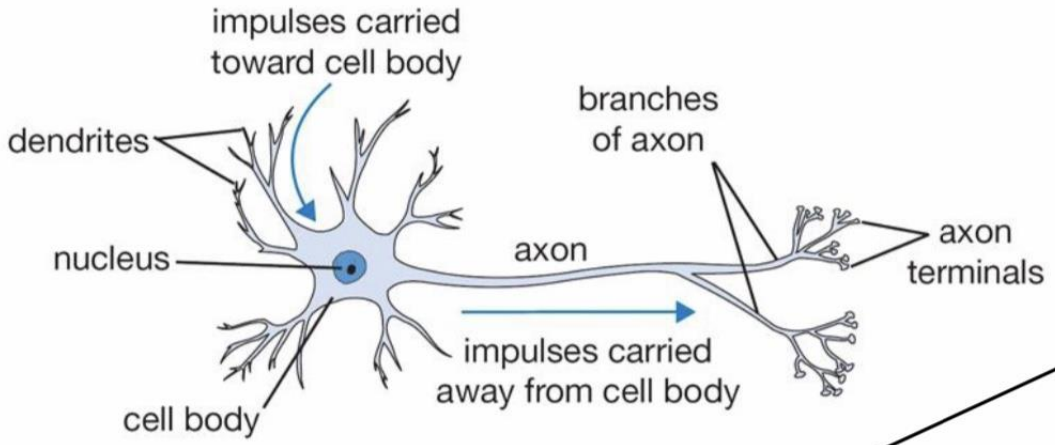


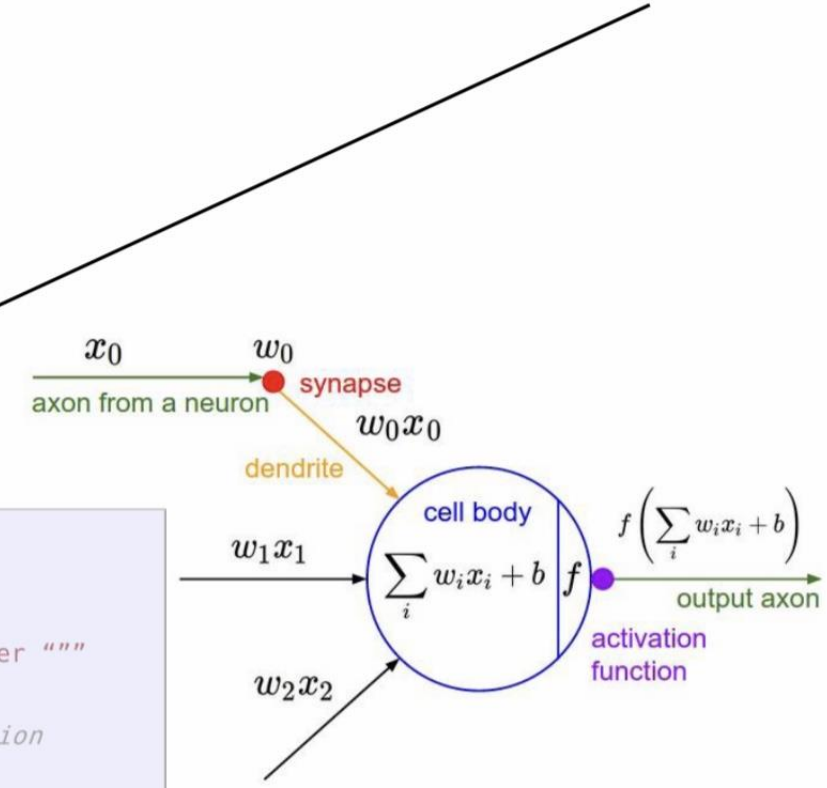
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Sigmoid activation function



Neurons and Neural Networks

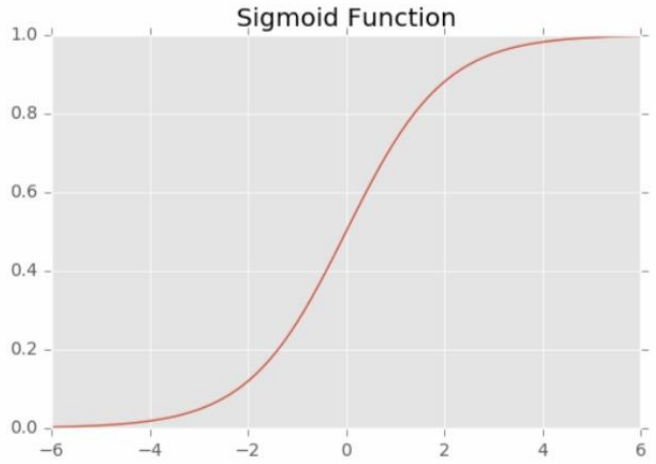





```
class Neuron:
    #...
    def neuron_tick(inputs):
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum))  # Sigmoid function
        return firing_rate
```

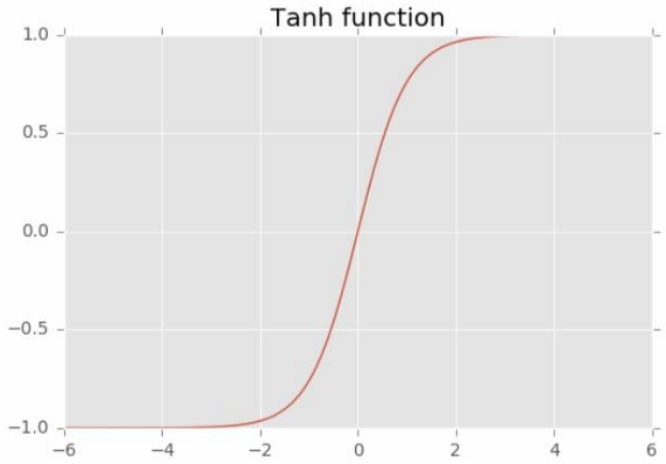
Activation Functions

sigmoid



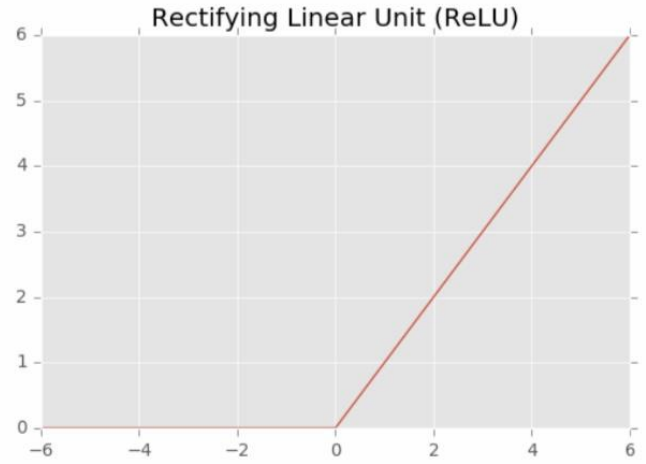
$$\sigma(x) = 1/(1 + e^{-x})$$

Tangant Hyperbolic



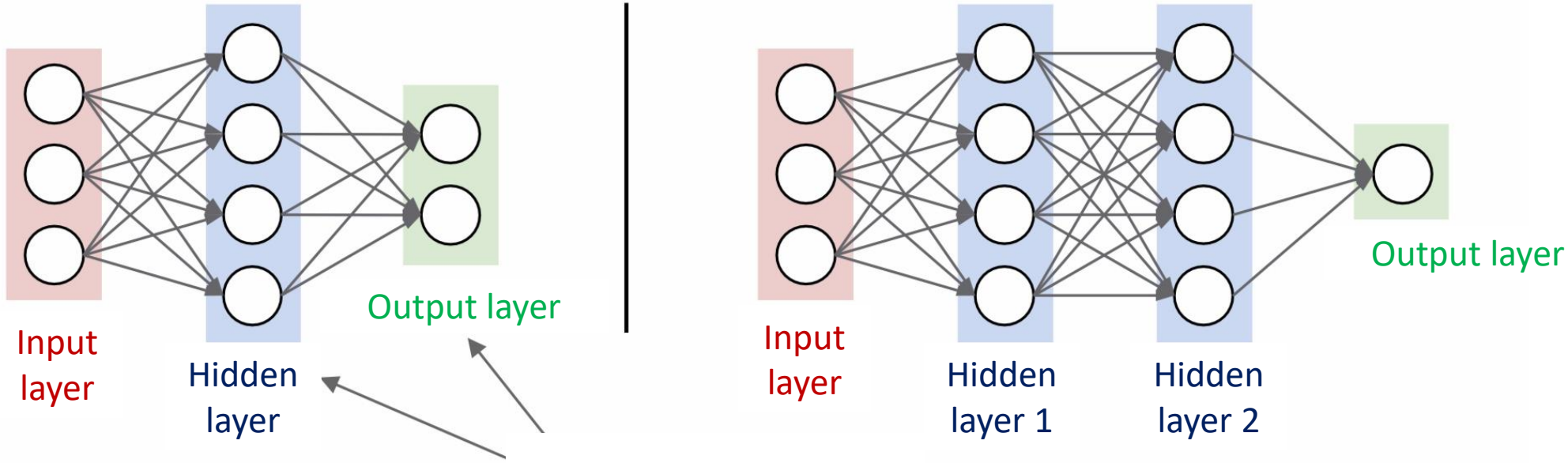
$$\tanh(x)$$

ReLU



$$\max(0, x)$$

Neural Networks: Architecture



Fully connected layers

2-layer neural network.
Neural network with 1 hidden layer

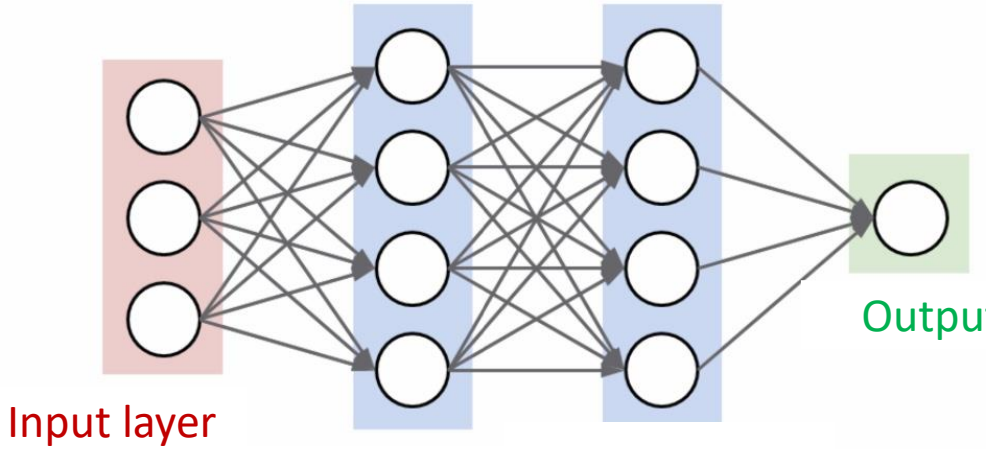
3-layer neural network.
Neural network with 2 hidden layer

Forward Calculation in a Neural Network

```
class Neuron:

    #...

    def neuron_tick(inputs):
        """ assume inputs and weights are 1-D numpy arrays and bias is a number """
        cell_body_sum = np.sum(inputs * self.weights) + self.bias
        firing_rate = 1.0 / (1.0 + math.exp(-cell_body_sum))  # Sigmoid function
        return firing_rate
```

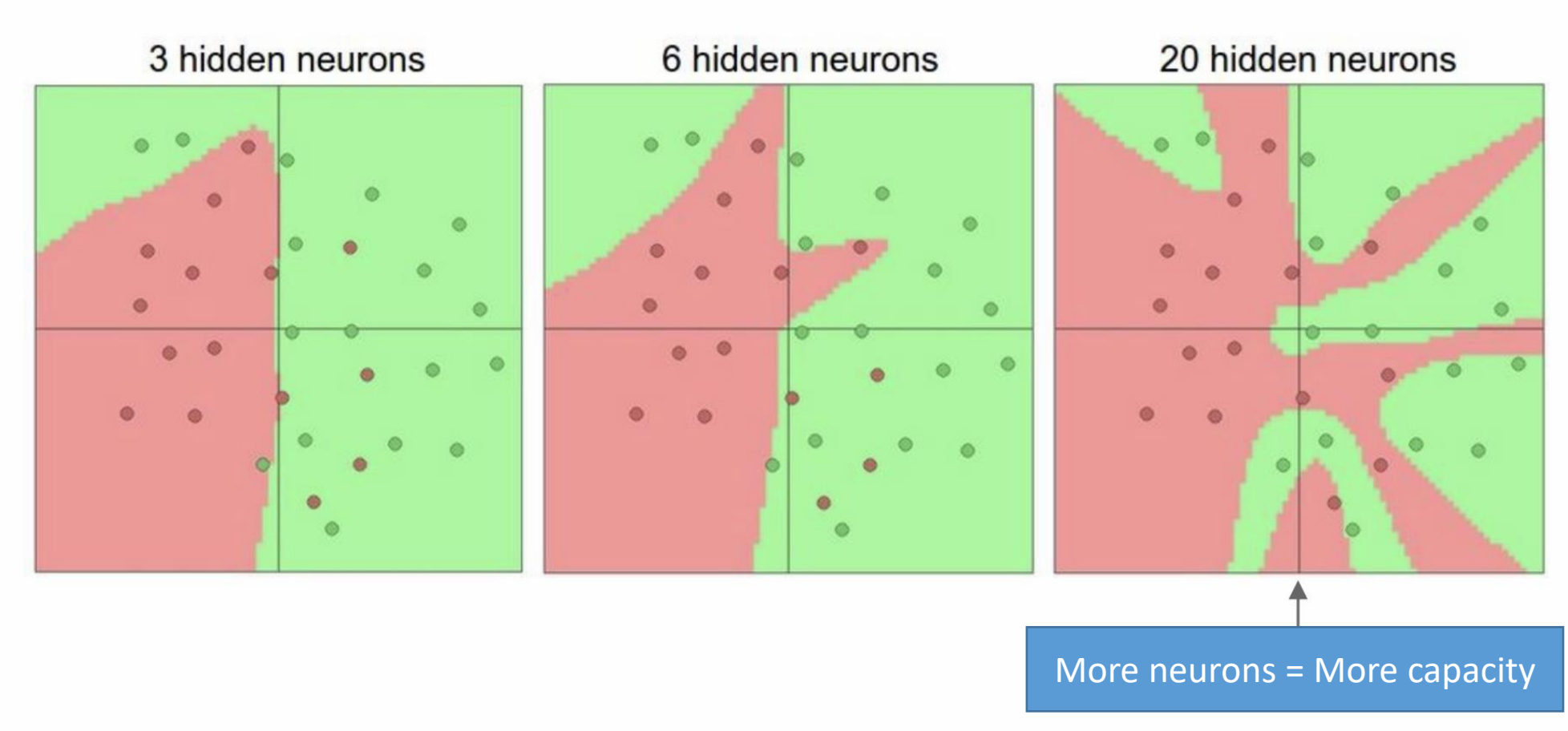
- Note: calculations of a complete layer of neurons can be efficiently implemented. (vector implementation)

Forward Calculation in a Neural Network

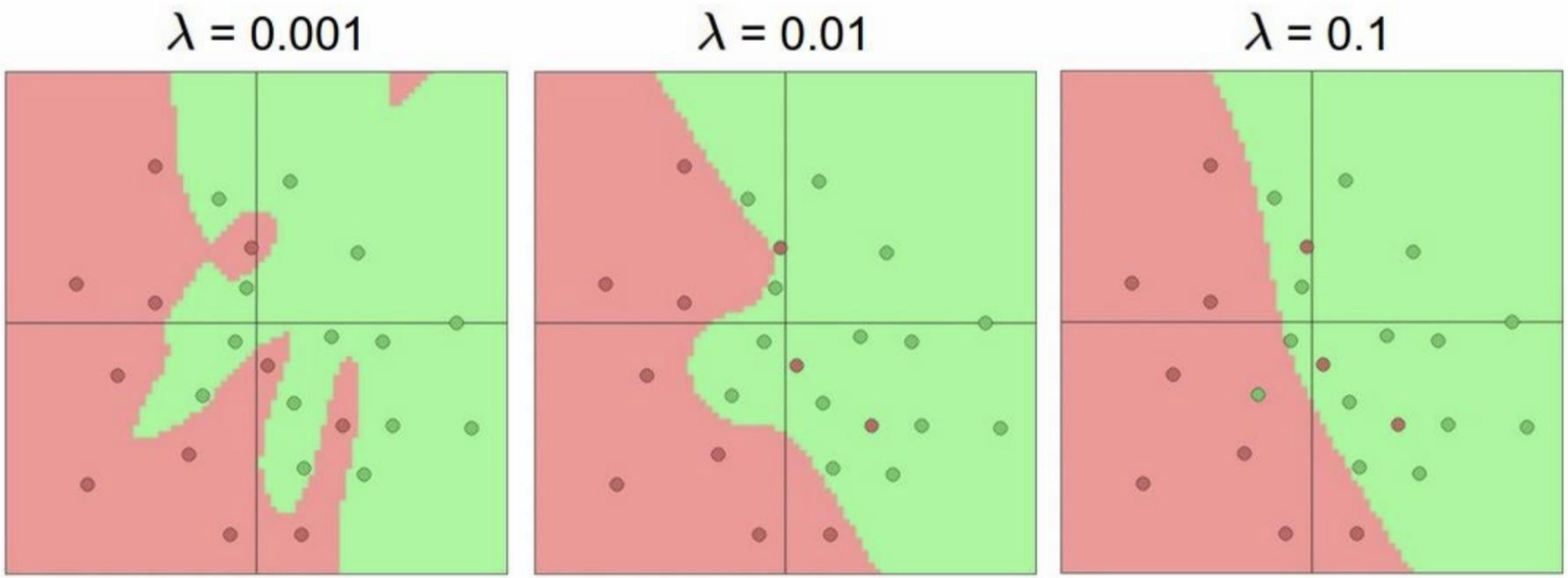



```
# forward pass of a 3-layer neural network:
f = lambda x: 1.0 / (1.0 + np.exp(-x))  # activation function (use sigmoid)
x = np.random.randn(3, 1)      # random input vector of three numbers (3x1)
h1 = f(np.dot(W1, x ) + b1)    # calculate first hidden layer activations (4x1)
h2 = f(np.dot(W2, h1)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

Determine the number and size of layers



Determine the number and size of layers



Do not use the size of the neural network as a regularizer and use a more robust method instead.

L2-Norm

Conclusion

- In a neural network, we place the neurons in a fully connected layer.
- Using the layer concept allows us to use the efficient implementation in vector form (such as matrix multiplication).
- Artificial neurons are very simplified models of biological neurons.
  - In fact, neural networks are not neural at all!
- The bigger the size of a neural network, the better:
  - Provided that a strong regulator is used to regulate the weights.

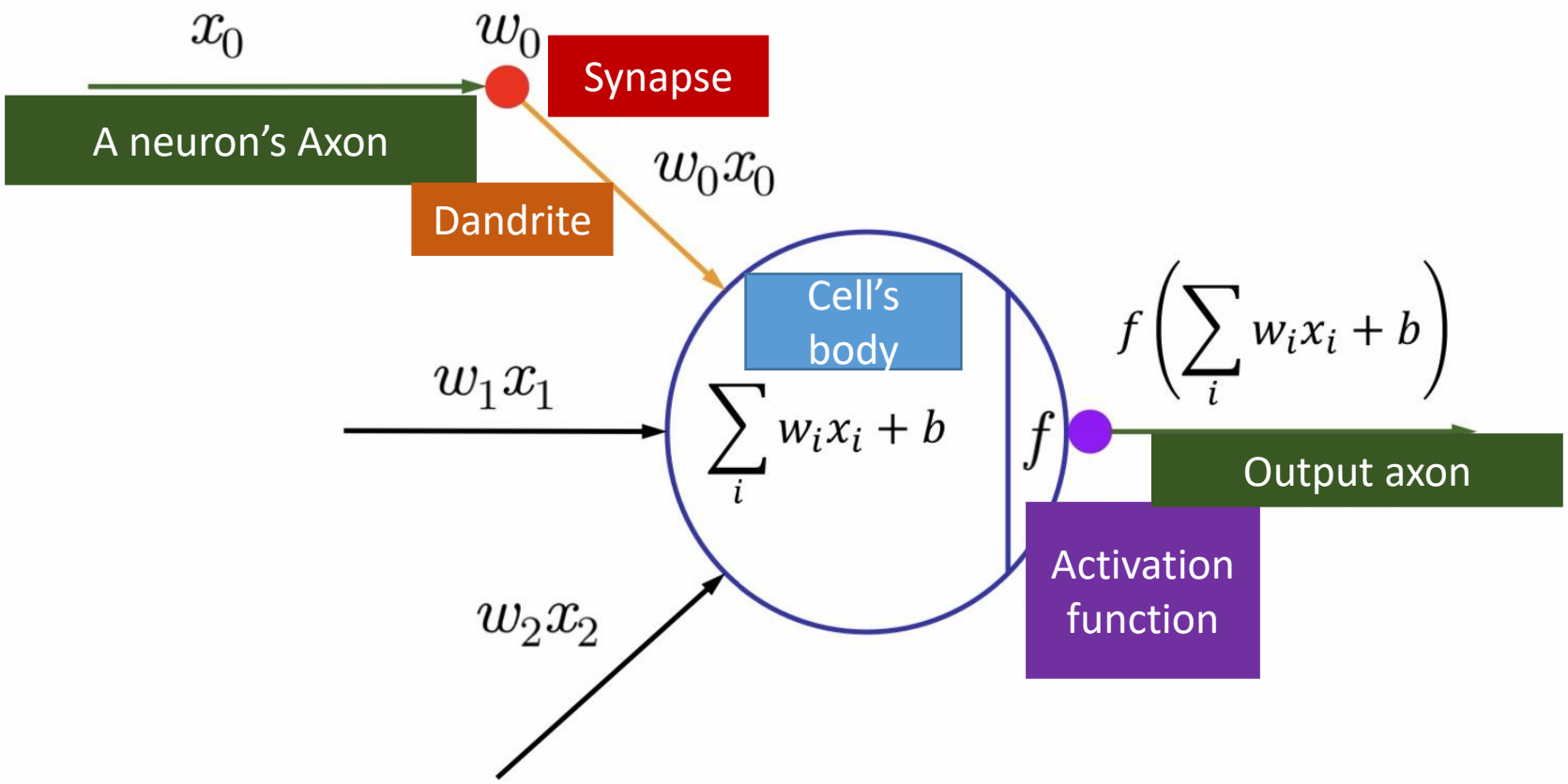
Neural Networks: Training

Neural Networks' training

- (1) Startup
  - Activity functions, preprocessing, initialization to weights, regularization, gradient checking
- (2) Training
  - Development stages of the learning process
  - How to update parameters, optimize hyperparameters
- (3) Evaluation

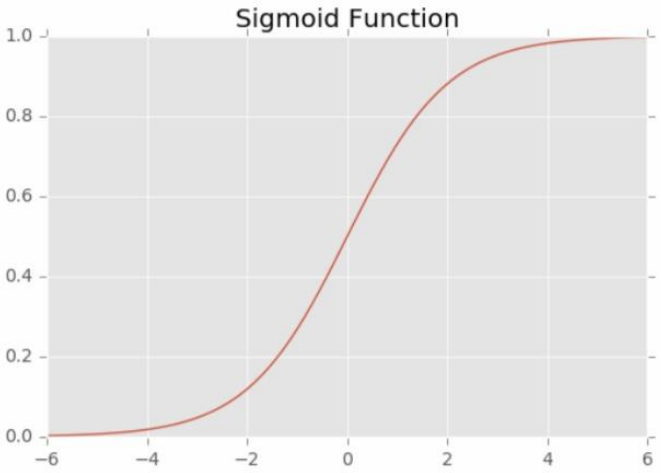
Cost Functions

Cost Functions



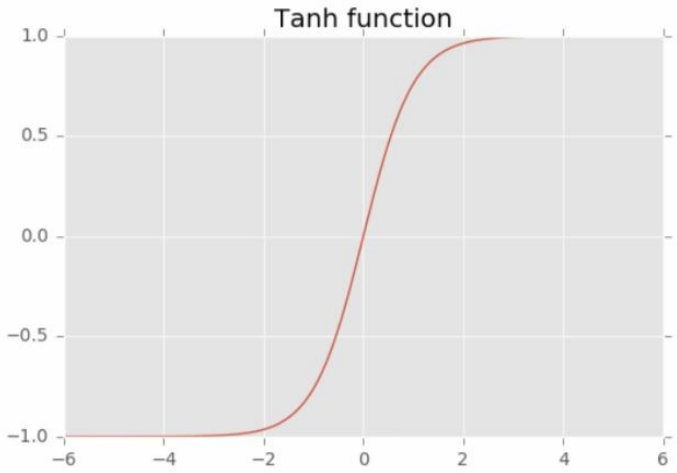
Cost Functions

Sigmoid



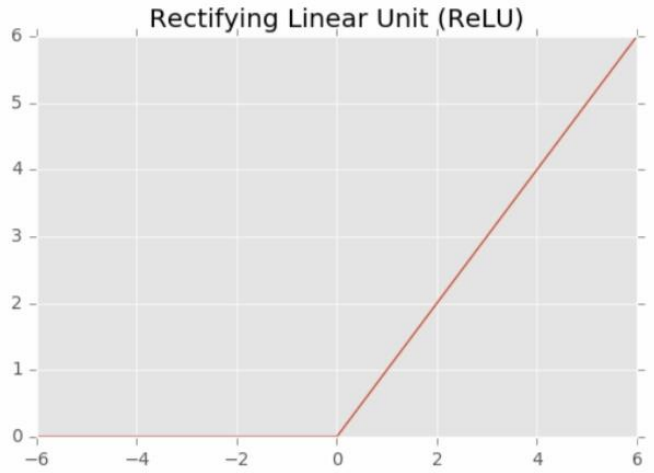
$$\sigma(x) = 1/(1 + e^{-x})$$

Tangant hyperbolic



$$\tanh(x)$$

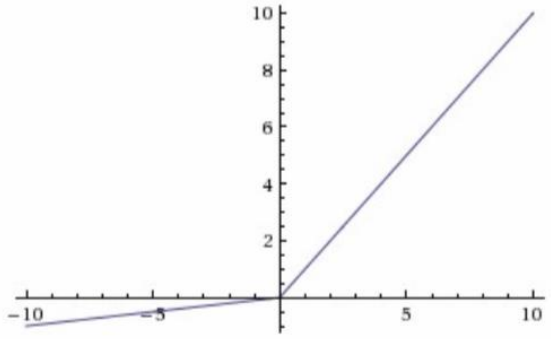
ReLU



$$\max(0, x)$$

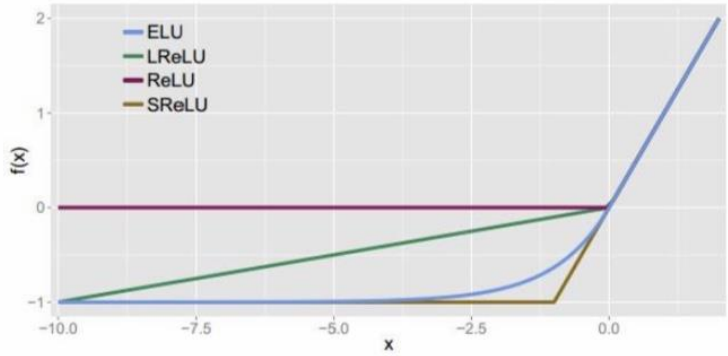
Cost Functions

Leaky ReLU



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

ELU



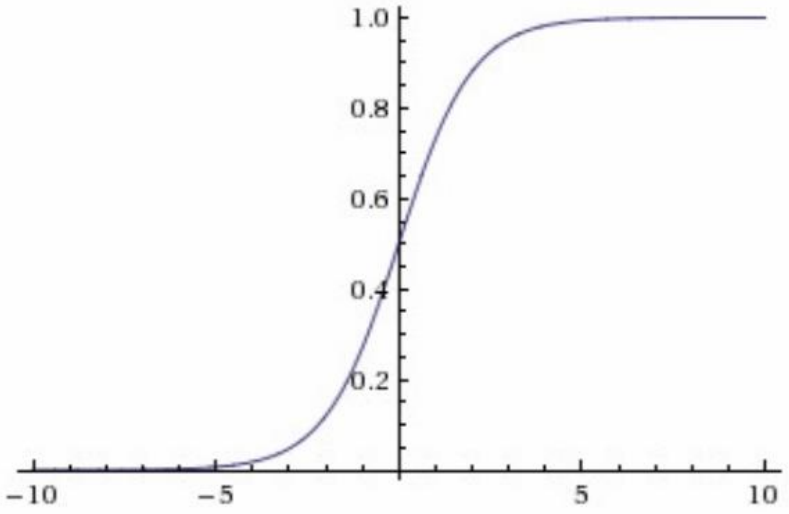
$$f(x) = \begin{cases} x, & x > 0 \\ \alpha(\exp(x) - 1), & x \geq 0 \end{cases}$$

Cost Functions

- Sigmoid activity function:
  - Mapping numbers to the interval [0, 1]
  - Historically popular in the history of neural networks

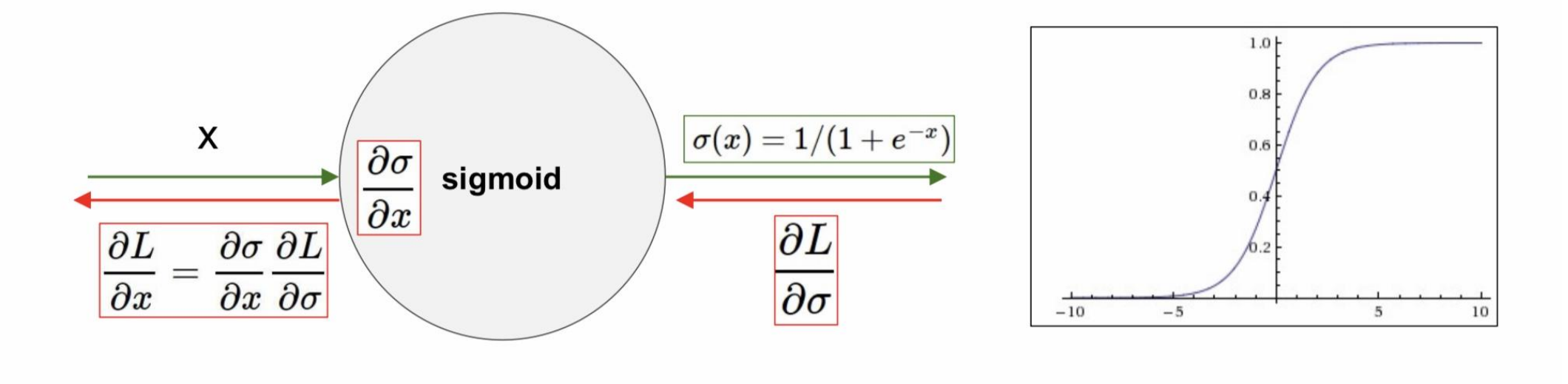
- Difficulties:
  - 1. Saturated neurons destroy the gradient.
  - 2. The outputs of the sigmoid function do not have a zero mean.
  - 3. Empowerment is a relatively expensive process.

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



Sigmoid function

Cost Functions



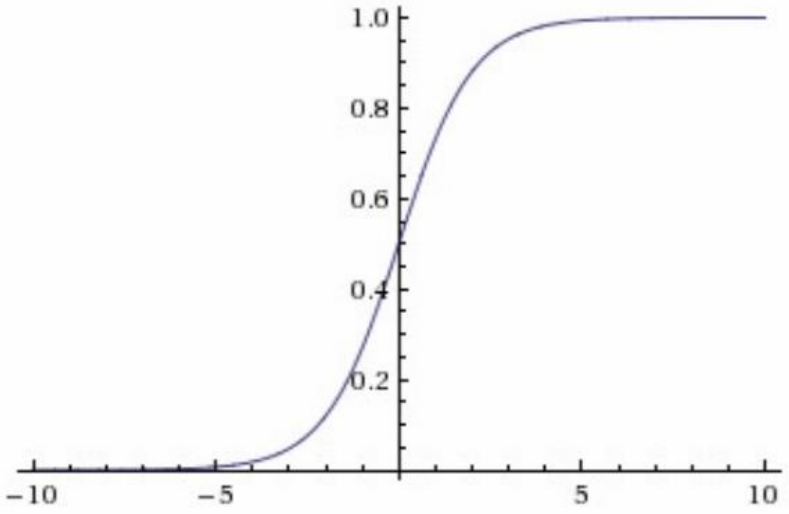
- What happens if x = -10?
- What happens if x = 0?
- What happens if x = +10?

Cost Functions

- Sigmoid activity function:
  - Mapping numbers to the interval [0, 1]
  - Historically popular in the history of neural networks

- Difficulties:
  - 1. Saturated neurons destroy the gradient.
  - 2. The outputs of the sigmoid function do not have a zero mean.
  - 3. Empowerment is a relatively expensive process.

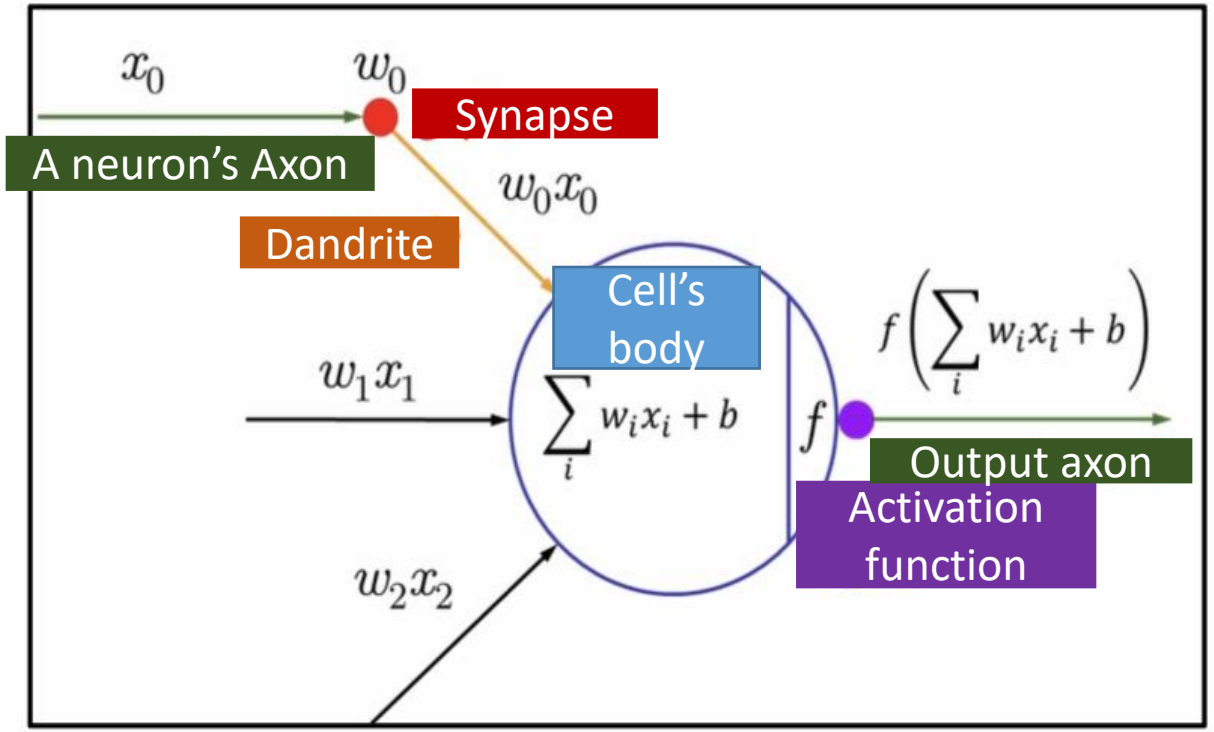
$$\sigma(x) = 1/(1 + e^{-x})$$



Sigmoid function

Cost Functions

- What happens if the input value of a neuron (x) is always positive?

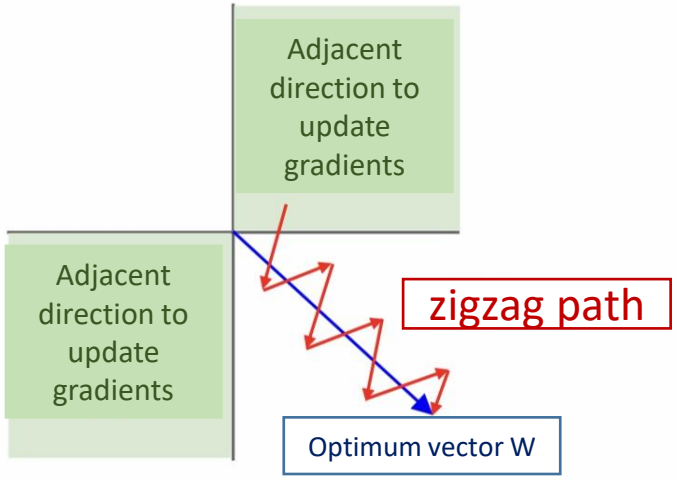


$$f\left(\sum_i w_i x_i + b\right)$$

- What can be said about gradients with respect to W?

Cost Functions

- What happens if the input value of a neuron (x) is always positive?



$$f\left(\sum_i w_i x_i + b\right)$$

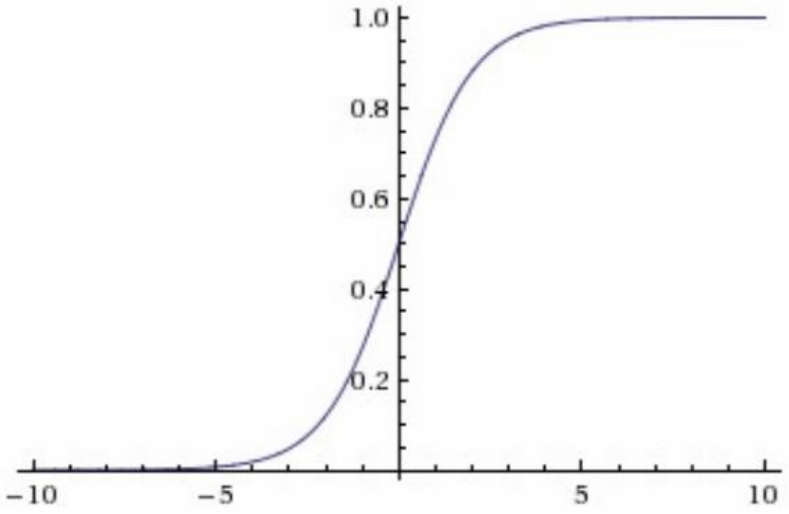
- What can be said about gradients with respect to W?
  - All will be positive or all negative!
  - For this reason, we want the data to have a zero mean.

Cost Functions

- Sigmoid activity function:
  - Mapping numbers to the interval [0, 1]
  - Historically popular in the history of neural networks

- Difficulties:
  - 1. Saturated neurons destroy the gradient.
  - 2. The outputs of the sigmoid function do not have a zero mean.
  - 3. Empowerment is a relatively
  - expensive process.

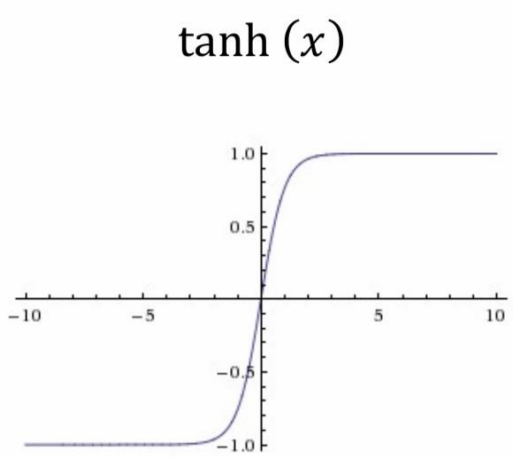
$$\sigma(x) = 1/(1 + e^{-x})$$



Sigmoid function

Cost Functions

- Tangent hyperbolic activation function
  - Mapping numbers to the interval [-1, 1]
  - Has a mean of zero
  - Still saturated neurons eliminate the gradient.



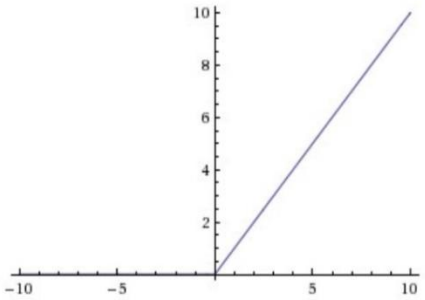
Tangent hyperbolic

[LeCun et al., 1991]

Cost Functions

- Does not saturate (in positive areas)
- It is very computationally efficient.
- In practice, it converges much faster than sigmoid and hyperbolic tangent functions. (for example, 6 times)
- It does not have a mean of zero!

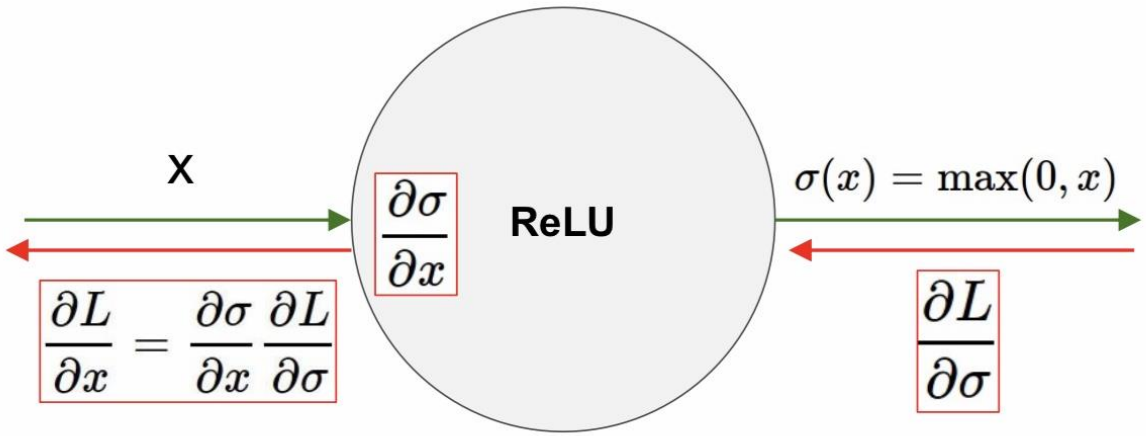
$$\max(0, x)$$

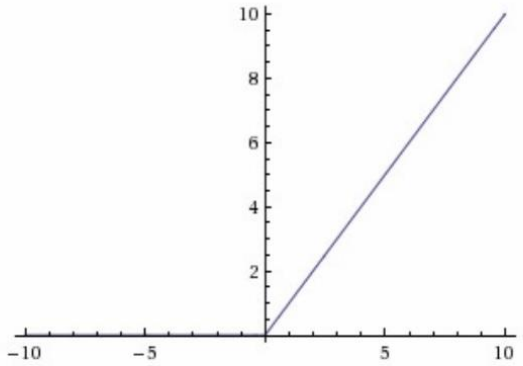


ReLU

[Krizhevsky et al., 2012]

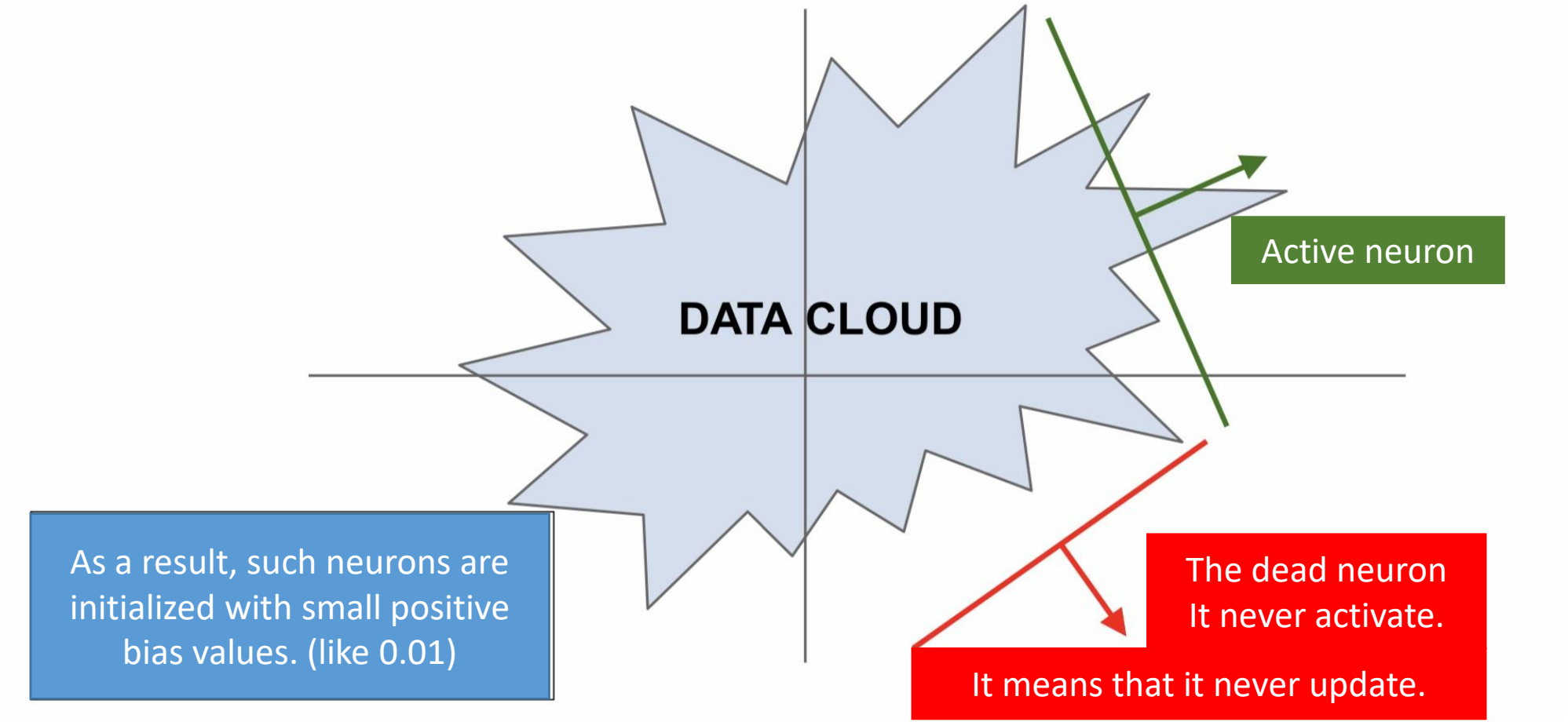
Cost Functions





- What happens if x = -10?
- What happens if x = 0?
- What happens if x = +10?

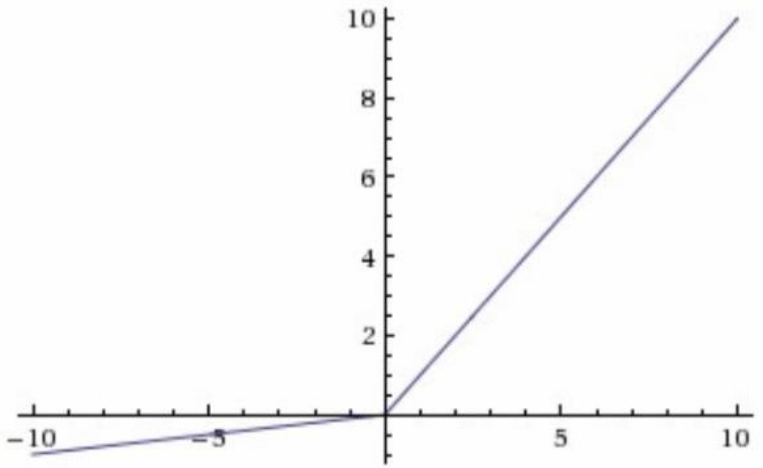
Cost Functions



Cost Functions

- Does not saturate (in positive areas)
- It is very computationally efficient.
- In practice, it converges much faster than sigmoid and hyperbolic tangent functions. (for example, 6 times)
- **Does not deactivate!**

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

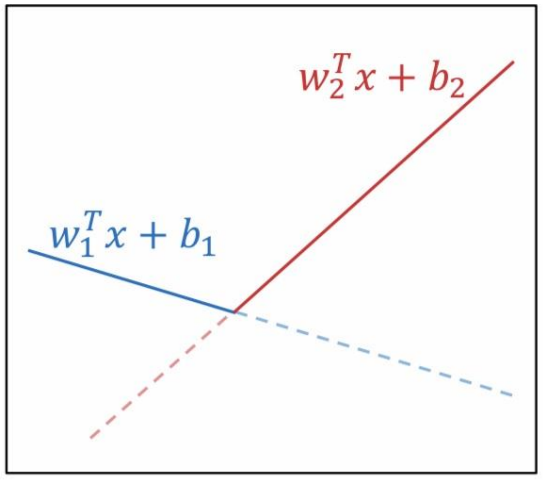


Leaky ReLU

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

Maxout activation function

- A piecewise linear function. (set of several linear functions)



- Generalizer of ReLU and LeakyReLU activity functions
- Linear behavior! Never saturated! It never deactivates!

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

- Problem: Doubling (many times) the number of parameters per neuron!

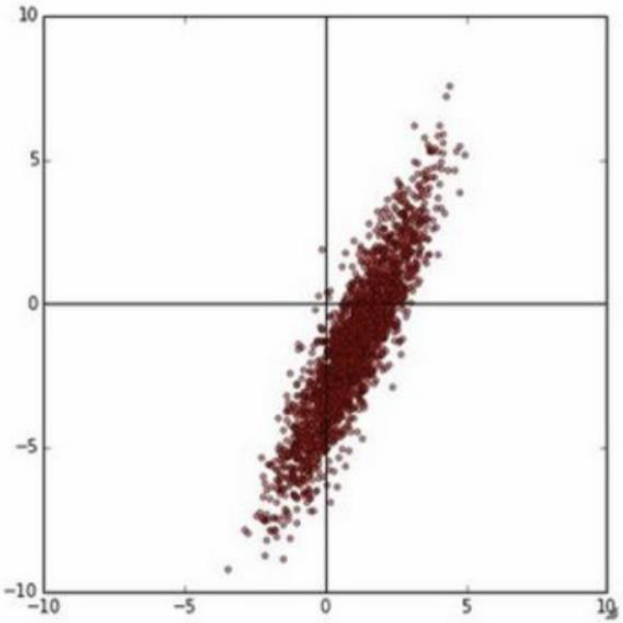
Choosing activation function

- in practice:
  - Use ReLU. Watch the learning rate.
  - Select Leaky ReLU and Maxout functions.
  - Also try the hyperbolic tangent function, but don't expect too much from it.
  - Do not use the sigmoid function.

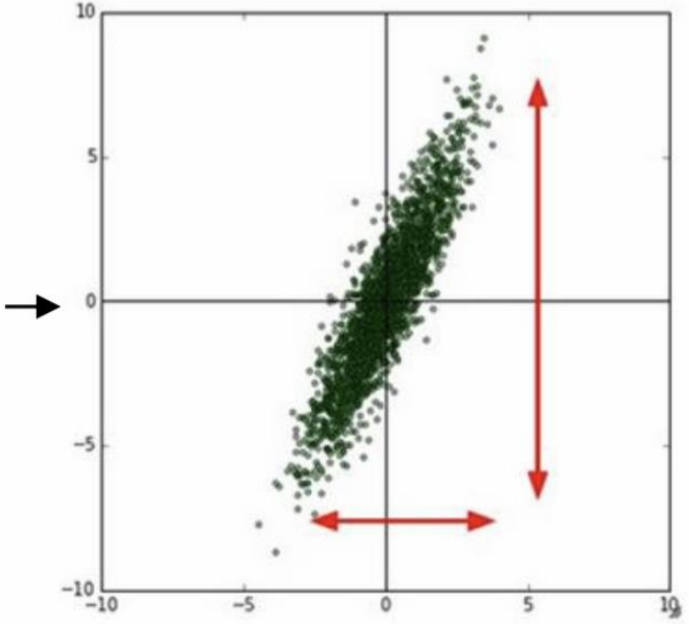
Data Preprocessing

Data Preprocessing

The main data

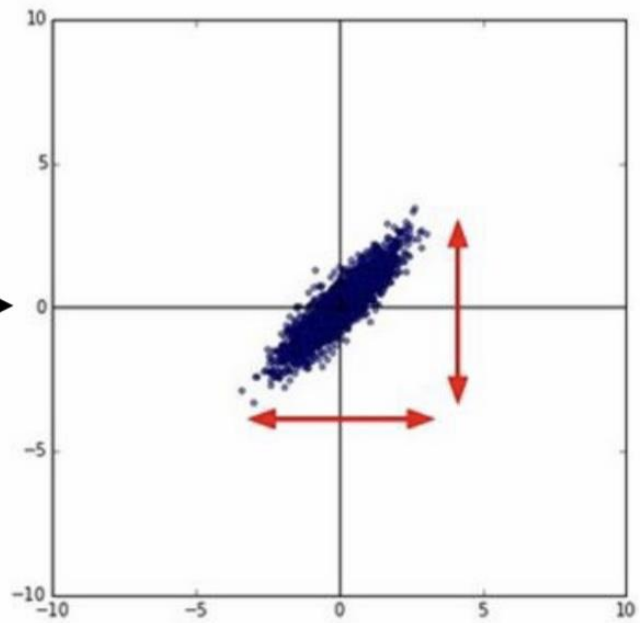


Data with zero mean




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

Normalized data



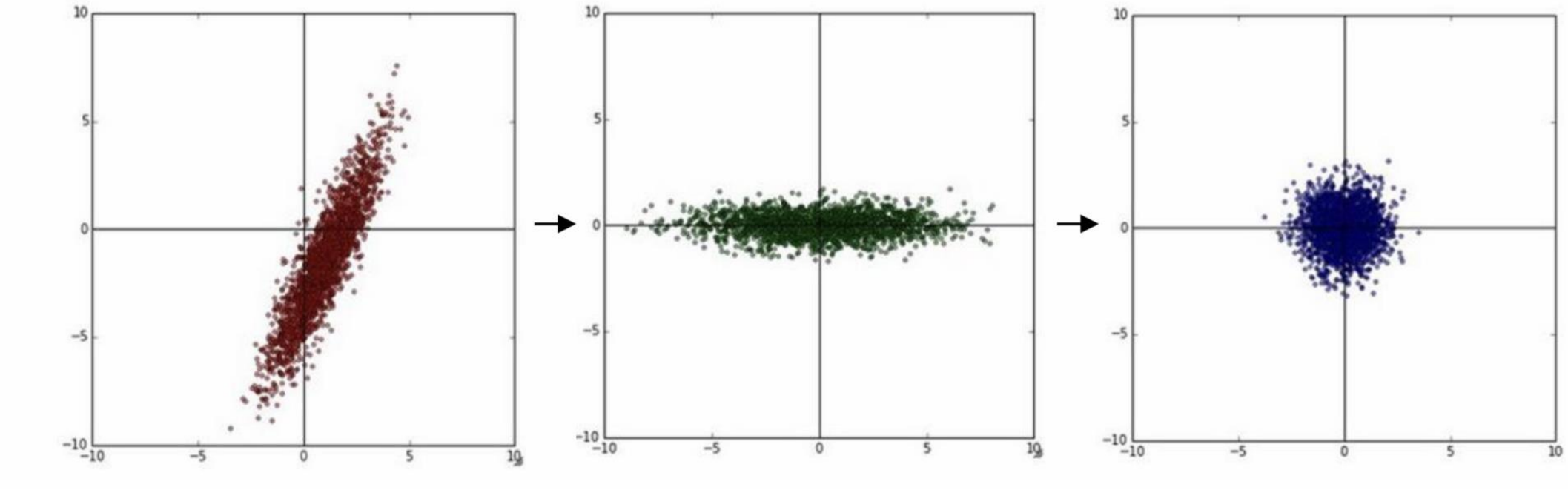

```
X /= np.std(X, axis=0)
```

Data Preprocessing

The main data

Uncorrelated data.

Whitened data.



Convert the covariance matrix to a diagonal matrix.

Convert the covariance matrix to an identity matrix.

Preprocessing for images

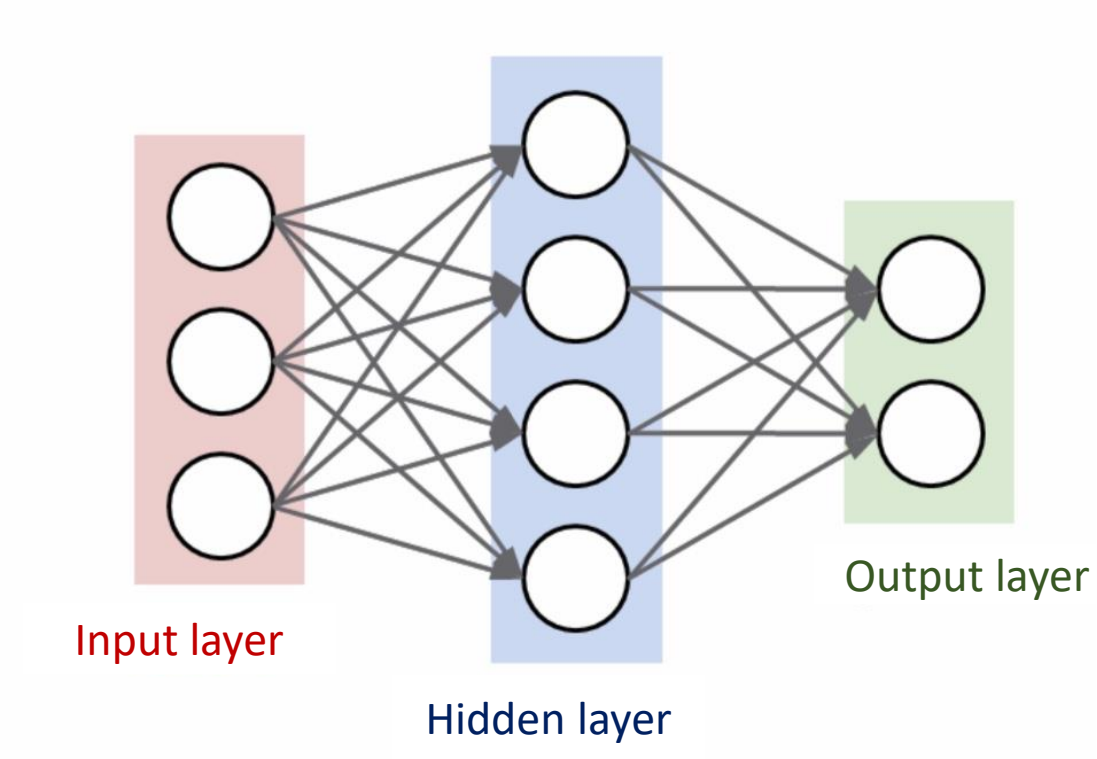
- In practice, only zeroing of the center is used for image data.
- As an example, consider the CIFAR-10 dataset images with dimensions [32, 32, 3]:
  - The first method: subtracting the average image from all images (such as AlexNet)
    - Average vector dimensions: [32, 32, 3]
  - Second method: subtracting the average for each color channel (such as VGGNet)
    - Average vector dimensions: 3 numbers

Variance normalization, decorrelation and whitening are not very common for image data.

Initializing the weights

Initializing the weights

- Question: What happens if all the weights are initialized to zero?



Initializing the weights

- First idea: small random numbers.

(Gaussian distribution with zero mean and 0.01 standard deviation)

```
W = 0.01 * np.random.randn(D, H)
```

- It is good for small networks, but for larger networks, it may lead to the heterogeneous distribution of the number of activities along different layers of the network!

Initializing the weights

- Example:
- A network with 10 layers,
- 500 neurons in each layer,
- Tanh activation function

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

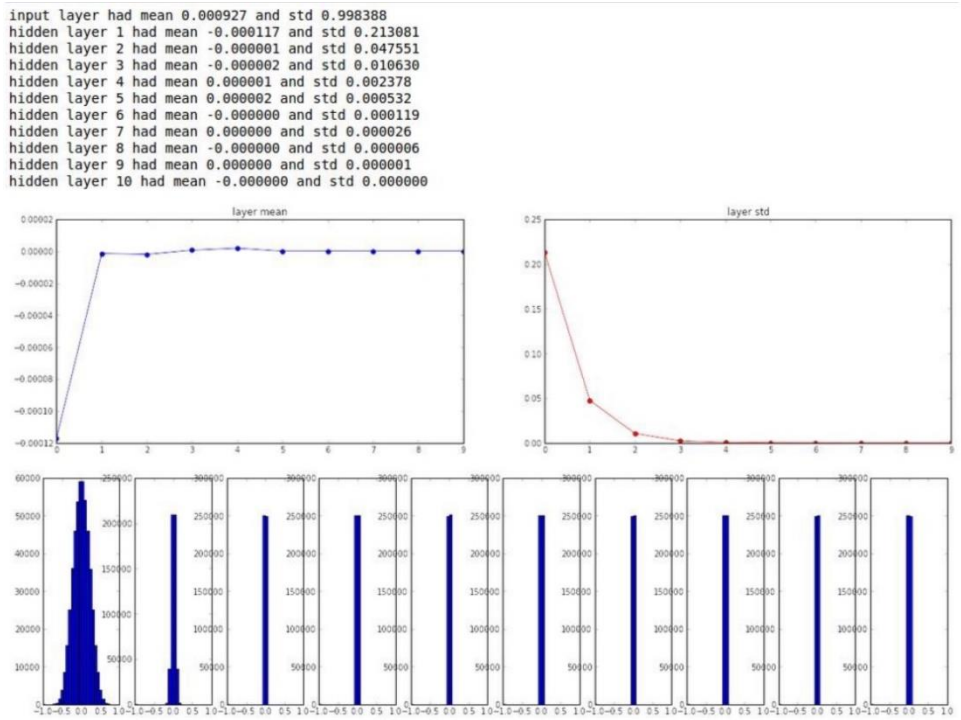
```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Initializing the weights

- The value of the neurons' activation becomes zero!

Question: How the value of the gradients will be during the backward computations?

Hint: In the computational graph, consider the nodes that perform the operation W * X.

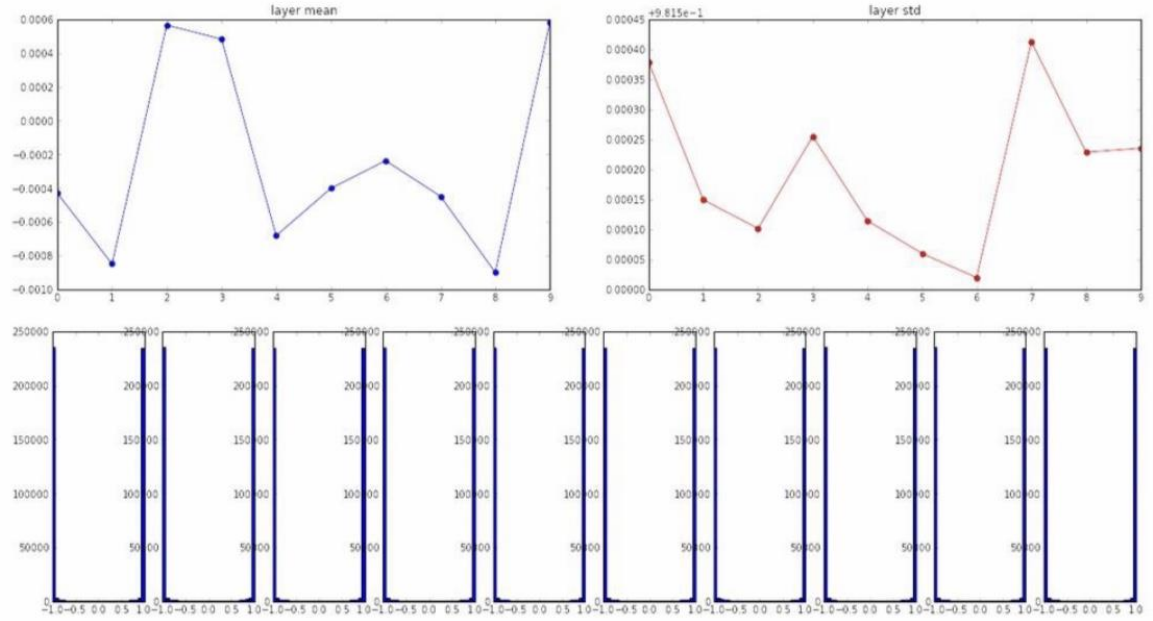



```
W = np.random.randn(fan_in, fan_out) * 0.01
```

Initializing the weights

- Almost all neurons are saturated. [-1 or +1].
- The value of all gradients will be zero.

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```



Weighting with large amounts →

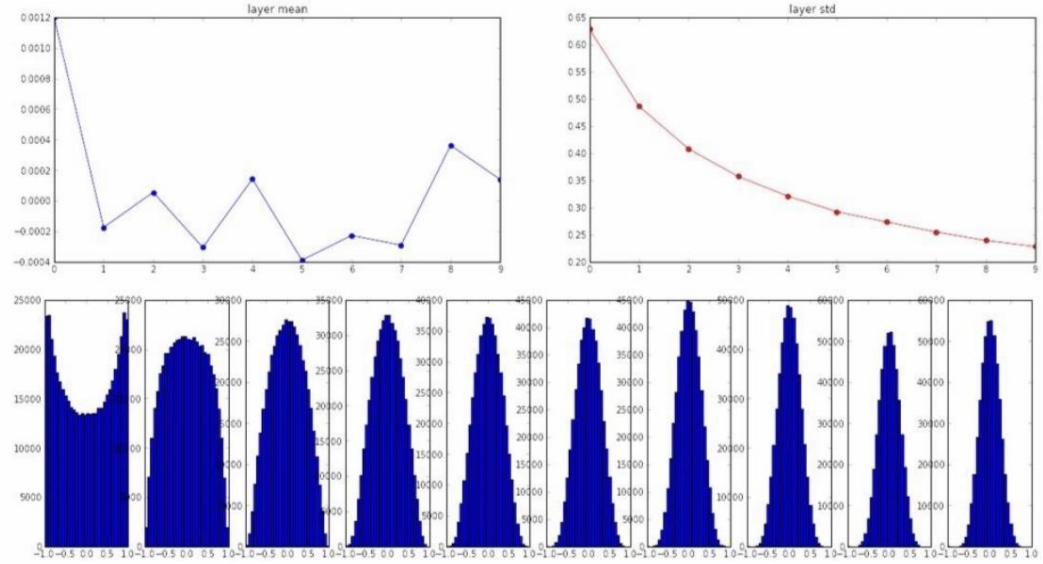

```
W = np.random.randn(fan_in, fan_out) * 1.0
```

Initializing the weights

- Xavier Weighting [Gloret et al. , 2010]

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

[Glo

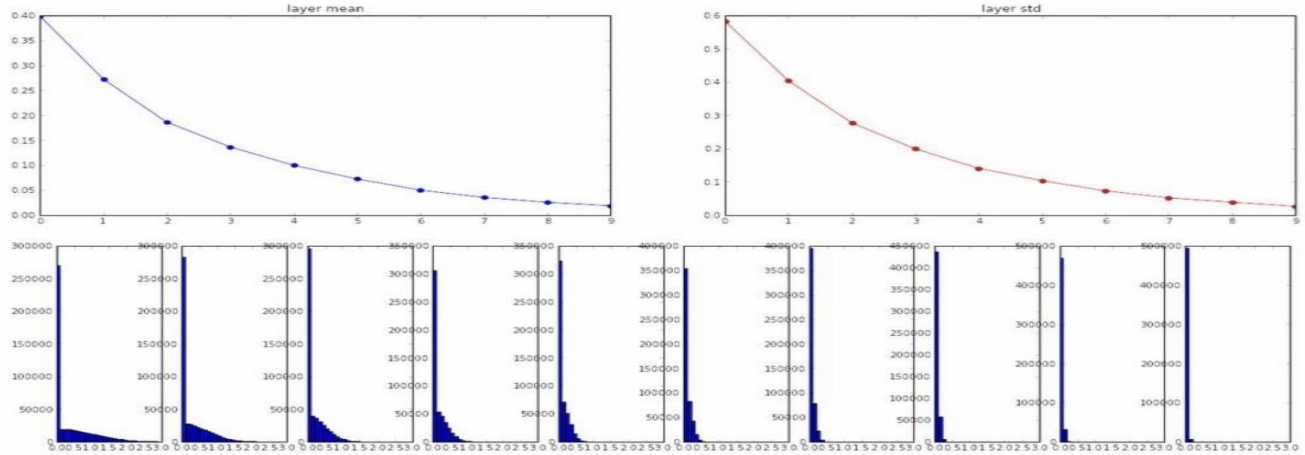



```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)
```

Initializing the weights

- Xavier Weighting [Gloret et al., 2010]
- But if you use the ReLU activation function, this method will no longer work properly!

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

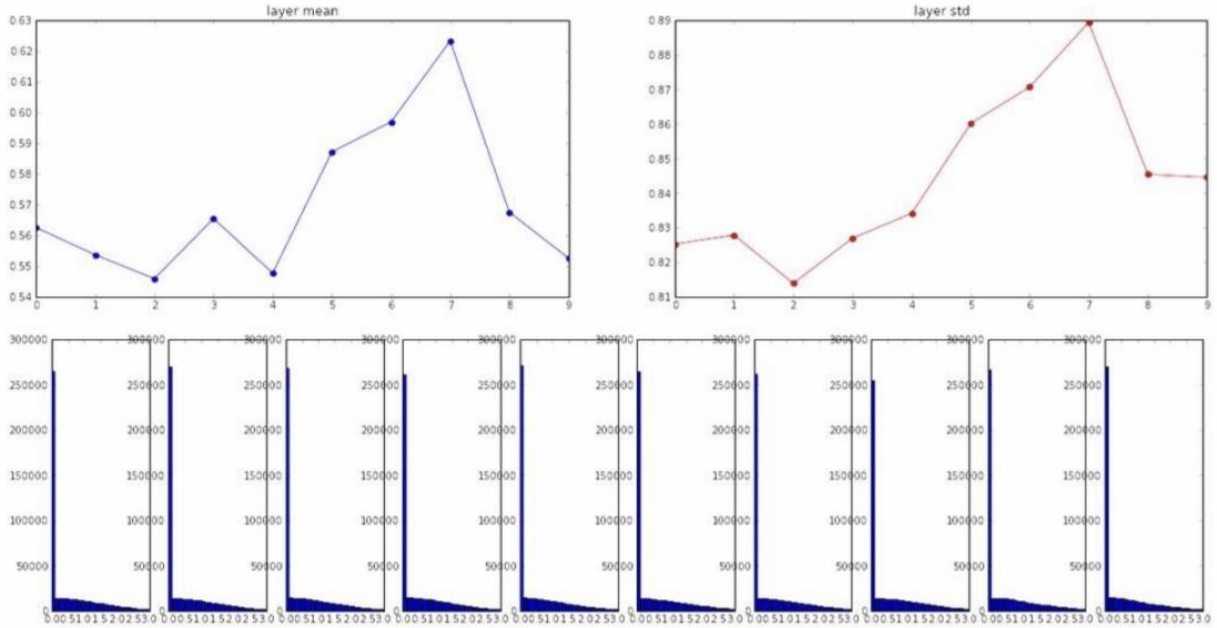


بیت
حل

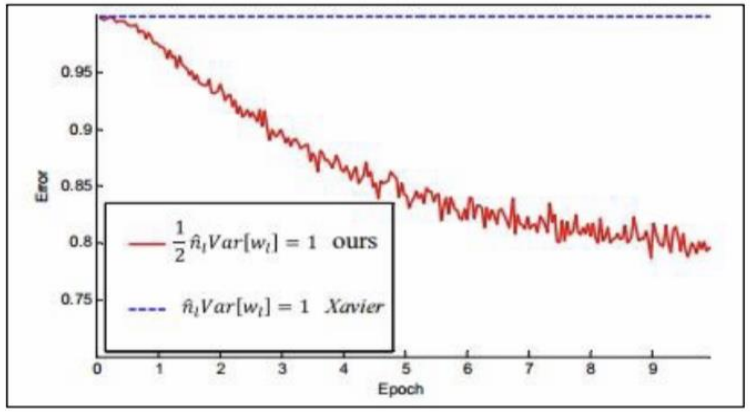
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)
```

Initializing the weights

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```



The better method. [He et al., 2015]
Note the factor of 1/2.

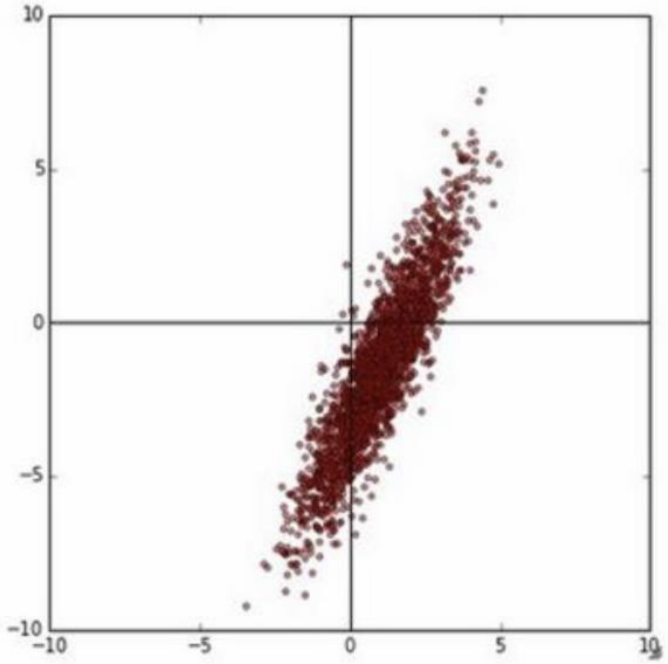



```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2)
```

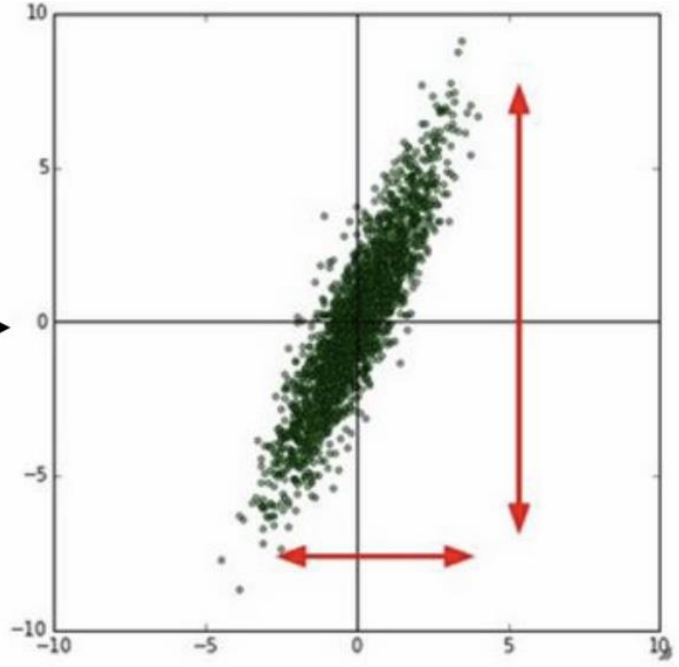
Development stages of the training process

The First step: Data Preprocessing

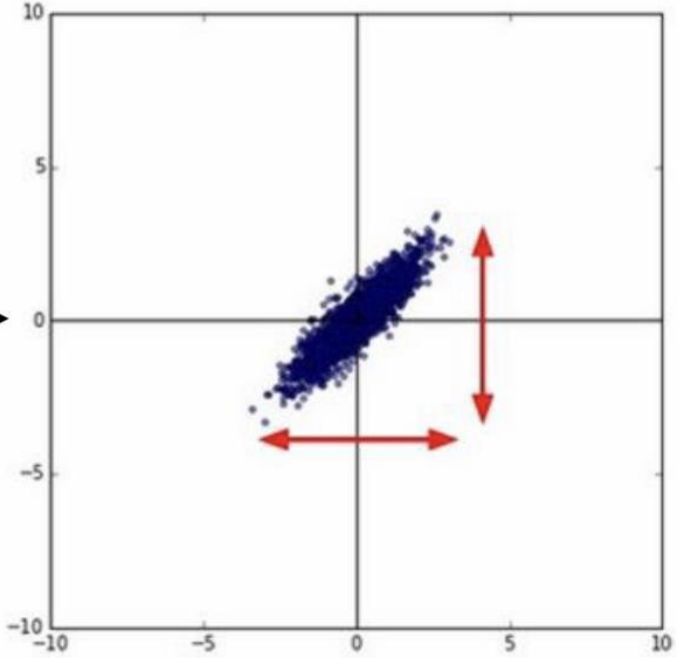
The main data



Data with zero mean



Normalized data

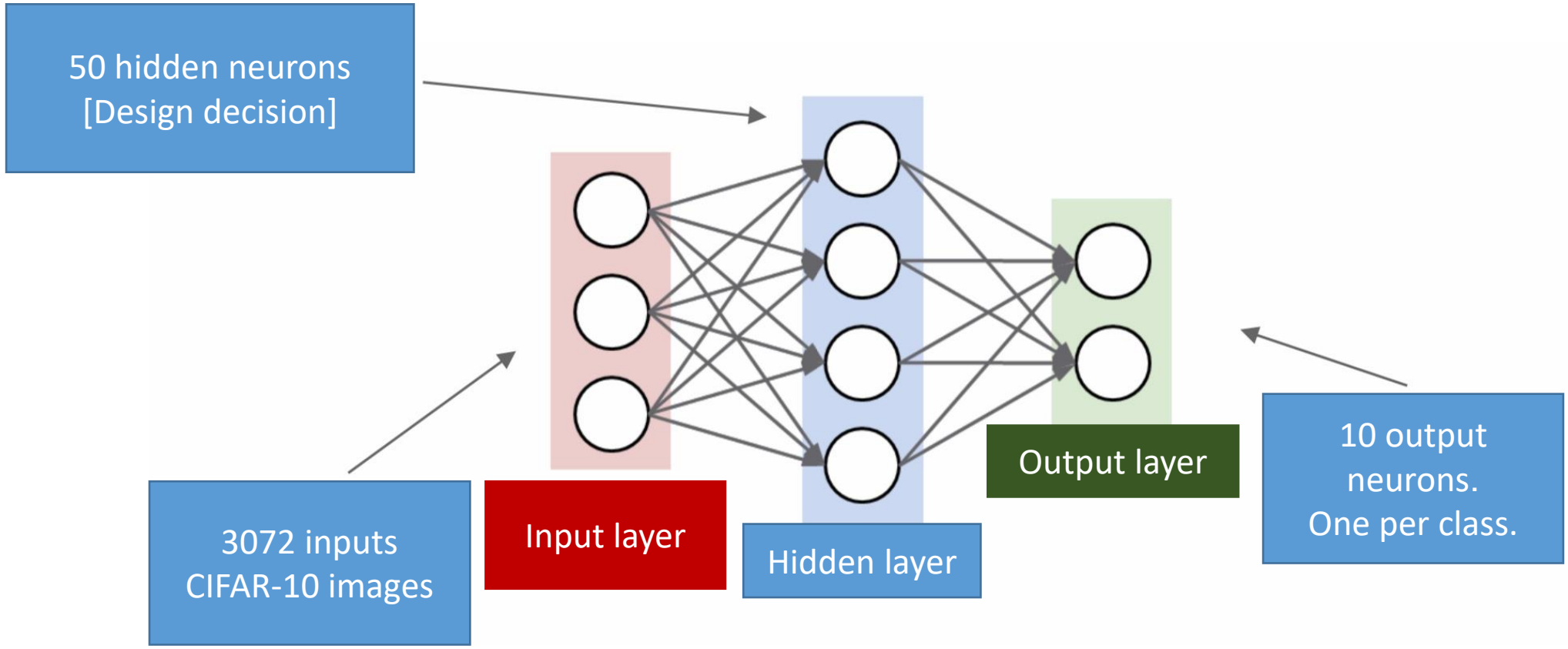


```
X -= np.mean(X, axis=0)
```

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

The second step: Architecture Selection

- Suppose we start with a hidden layer, containing 50 neurons.



The third step: checking the value of the cost function (without regularization)

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

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

Regularization disabling

```
2.30261216167
```

This value, for 10 classes, seems "right"!

Returning the value of the cost function and the gradient of all parameters

The third step: checking the value of the cost function (with regularization)

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

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

Enabling regularization

```
3.06859716482
```

The value of the cost function should increase.

Fourth step: preliminary training

- Note: make sure that training the network on a very small part of the training set causes overfitting!

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

- In the code snippet above:
  - The first 20 samples will be selected from the educational collection!
  - Regularization is disabled.
  - A simple version of gradient descent is used.

Fourth step: preliminary training

- Note: make sure that training the network on a very small part of the training set causes overfitting!

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

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-03
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-03
Finis
```

```
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.000000e-03
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.000000e-03
finished optimization. best validation accuracy: 1.000000
```

Very low cost function value, 100% training accuracy,

Fourth step: preliminary training

- Typically, we start with a very small regularization factor.
- We try to find a value for the learning rate, such that the value of the cost function decreases in each iteration.

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

Fourth step: preliminary training

- Typically, we start with a very small regularization factor.
- We try to find a value for the learning rate, such that the value of the cost function decreases in each iteration.

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

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

Non-decreasing cost function:
The learning rate is too small!

The value of the cost function changes very little.

Fourth step: preliminary training

- Typically, we start with a very small regularization factor.
- We try to find a value for the learning rate, such that the value of the cost function decreases in each iteration.

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

Now we try a large value like le6 for the learning rate.
What problems can occur?

Non-decreasing cost function:
The learning rate is too small!

Fourth step: preliminary training

- Typically, we start with a very small regularization factor.
- We try to find a value for the learning rate, such that the value of the cost function decreases in each iteration.

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

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

Cost function value: NaN
It almost always means that the learning rate value is too large.

Non-decreasing cost function:
The learning rate is too small!

The explosive growth of the cost function:
The learning rate is too large!

Fourth step: preliminary training

- Typically, we start with a very small regularization factor.
- We try to find a value for the learning rate, such that the value of the cost function decreases in each iteration.

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

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

The value of 3e-3 is still too large.
The cost function has explosive growth.

Non-decreasing cost function:
The learning rate is too small!

The explosive growth of the cost function:
The learning rate is too large!

As a result, the appropriate interval for the value of the learning rate is [le-5, le-3].

Optimization of hyperparameters: a cross-validation strategy

- Coarse-to-Fine Strategy: Start with a large search interval and then narrow this interval repeatedly.
  - Initial steps:
    - A small number of training courses to get a rough idea about the appropriate value of the hyperparameters.
  - next levels:
    - Longer runs, more detailed search (repeat if needed)
  - A tip for early detection of explosive growth in the cost function:
    - End the run whenever the amount of cost exceeds 3 times the initial cost.

Optimization of hyper parameters: a cross-validation strategy

- Example: Running a coarse search with 5 courses.

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

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

Optimization in logarithmic space

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

good

Optimization of hyper parameters: a cross-validation strategy

- Example: Running a coarse search with 5 courses.

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

Set the search range

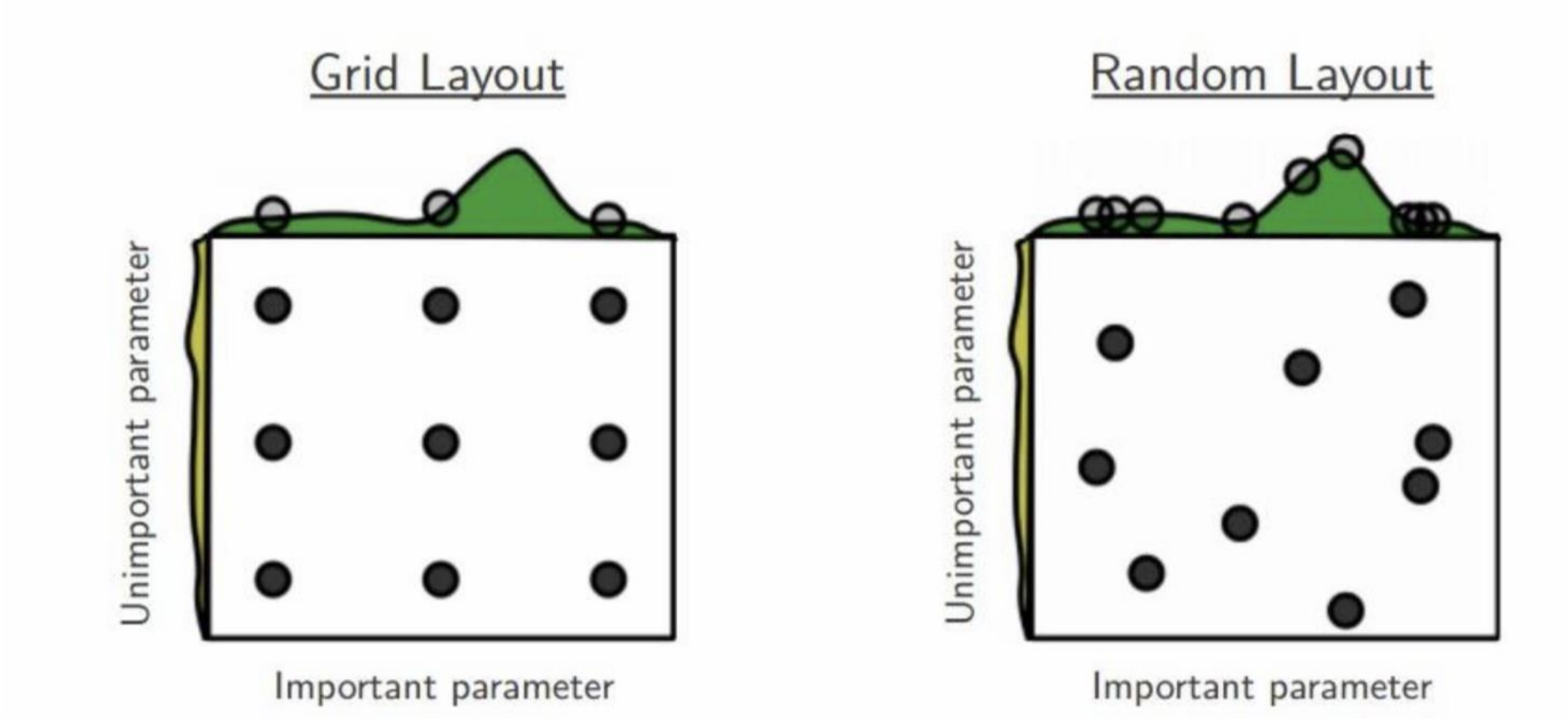
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

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

-53% for a two-layer network, with 50 hidden neurons, is a relatively good result!

But the best result is worrying. Why?

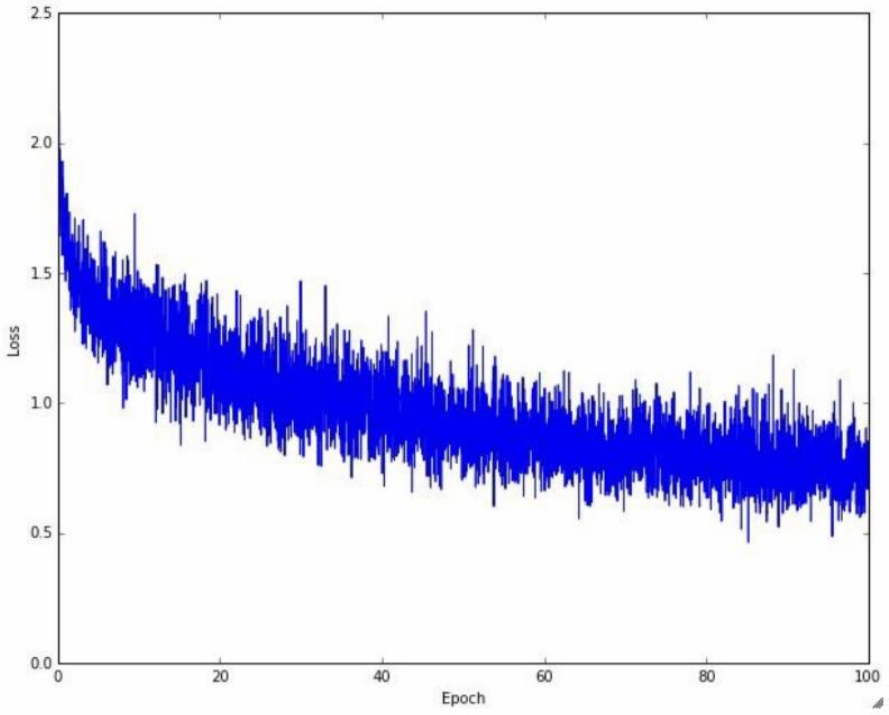
Optimization of hyper parameters: Random search or Grid search?

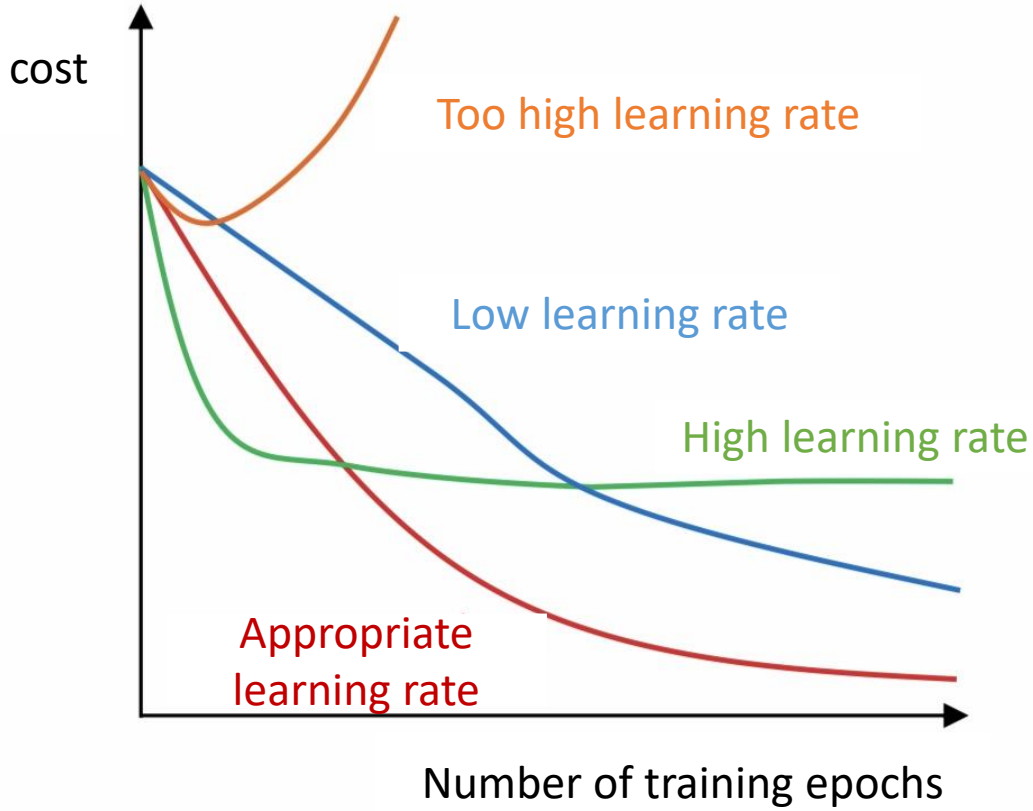


Optimization of hyper parameters

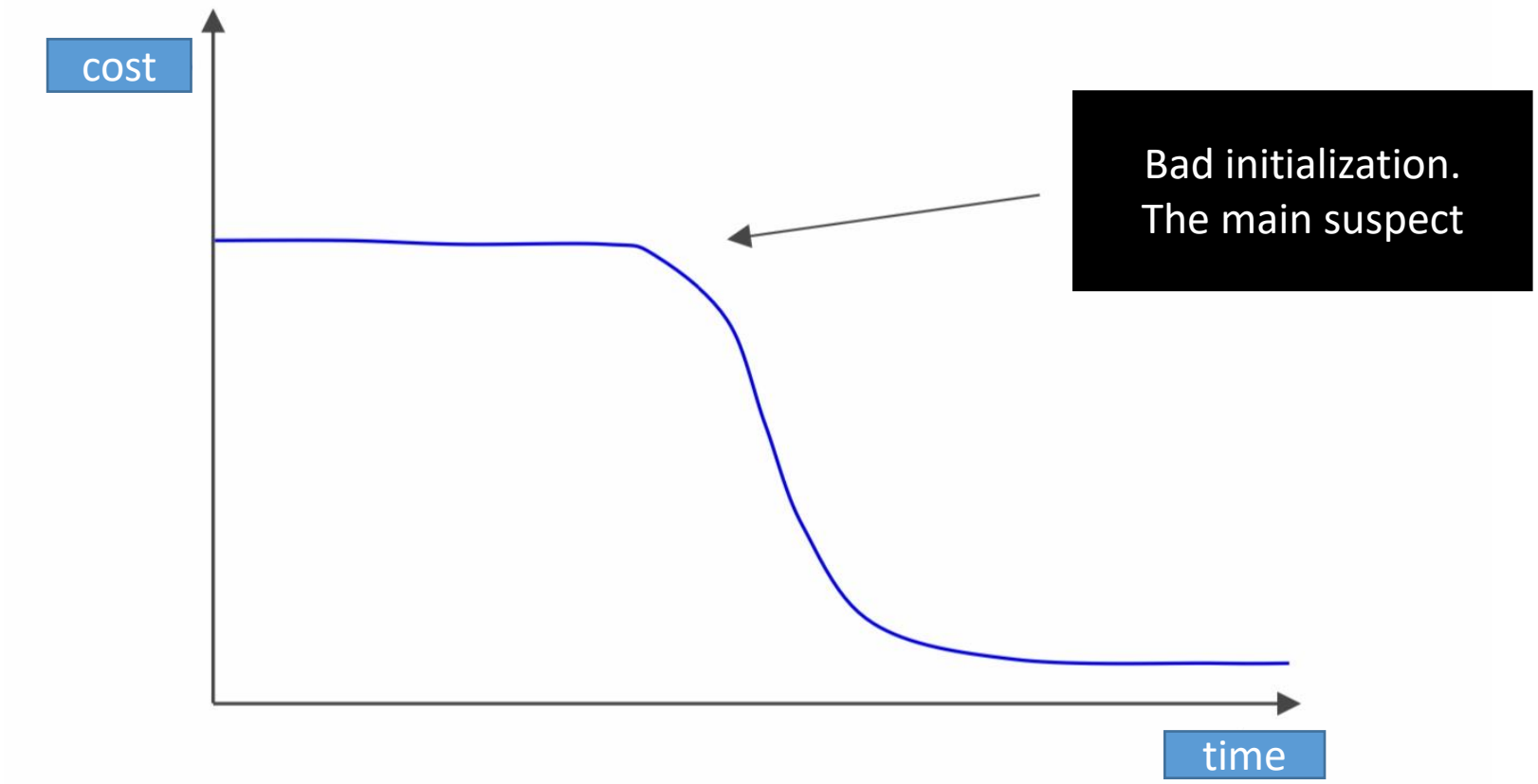
- hyper parameters that must be set:
  - Network structure (number and size of layers)
  - Learning rate, how to reduce it, update rule
  - How to regularize and its intensity

Monitoring and plotting the change of the cost function

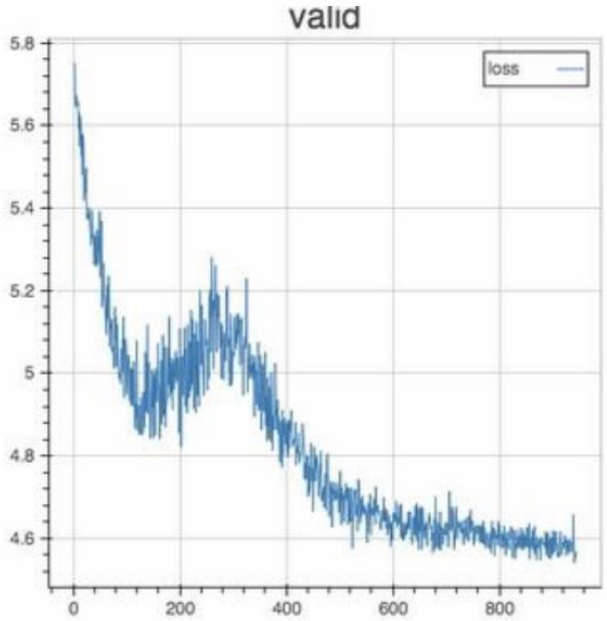


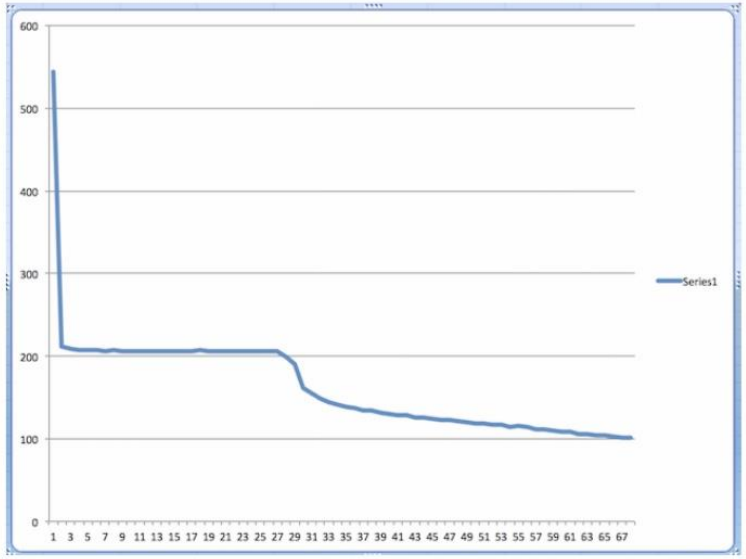


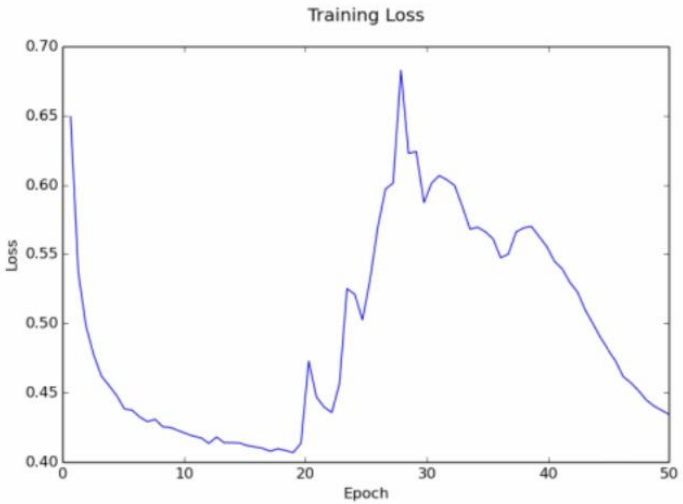
Monitoring and plotting the change of the cost function



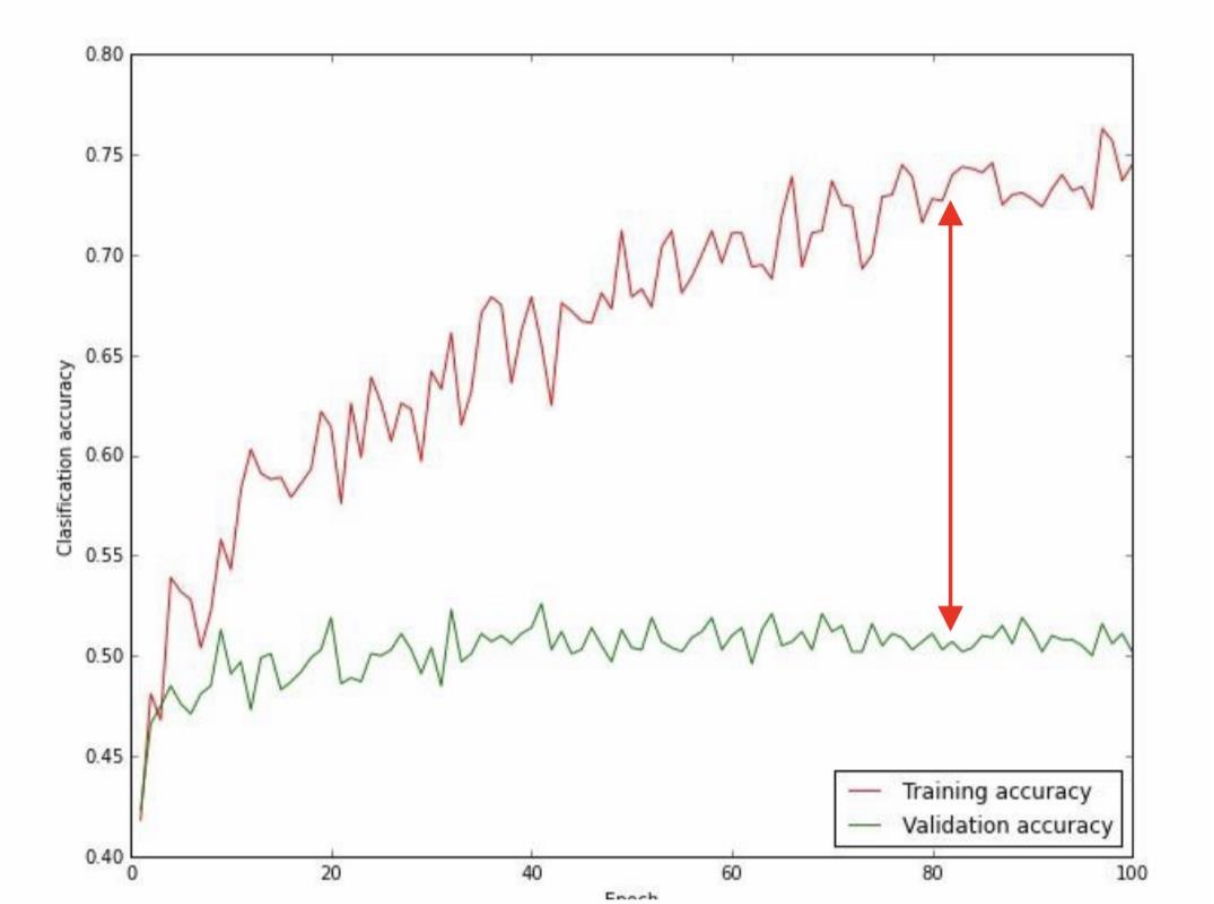
Monitoring and plotting the change of the cost function







Monitoring and plotting the change of the cost function



High difference = overfitting

☞ Increasing the regularization's intense?

No difference

☞ Increasing the model's capacity?

The ratio of the change of weights to the size of the weights

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

- The ratio of the update value to the weight value:

~0.0002 / 0.02 = 0.01

- A desirable ratio is around 0.001 or more.

Conclusion

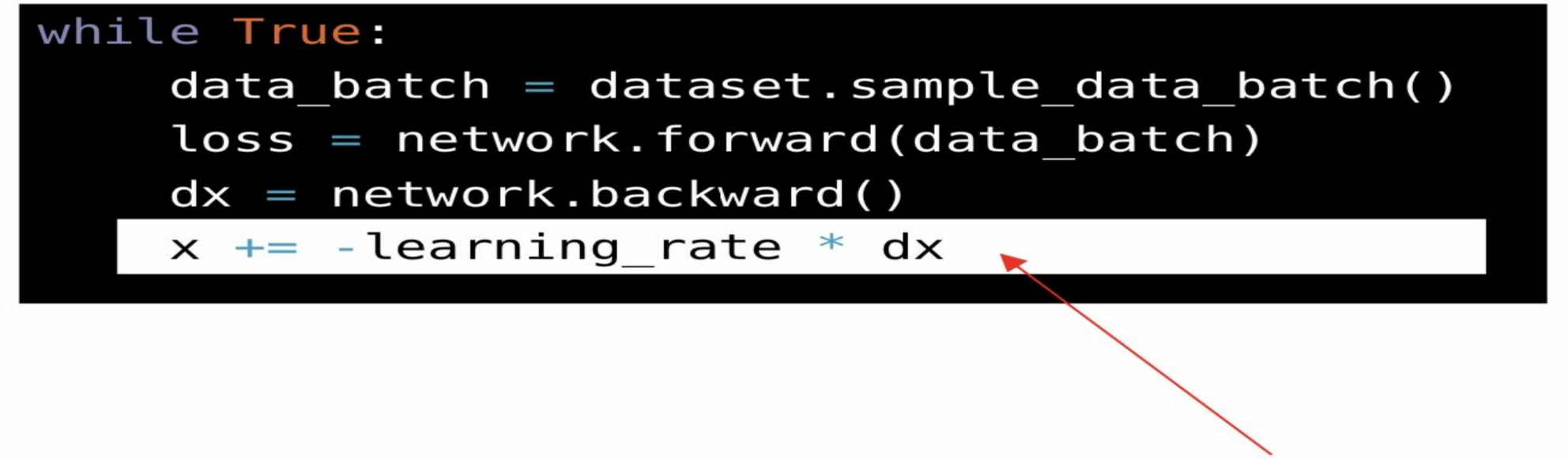
- Activation functions. [use ReLU]
- Data preprocessing. [for image: mean subtraction]
- Weights initializations. [use Xavier method]
- Development stages of the education process. [pre-processing, architecture selection, initial training, super-parameters optimization]
- Optimization of hyperparameters [random search in logarithmic space]

Table of Contents

Updating parameters

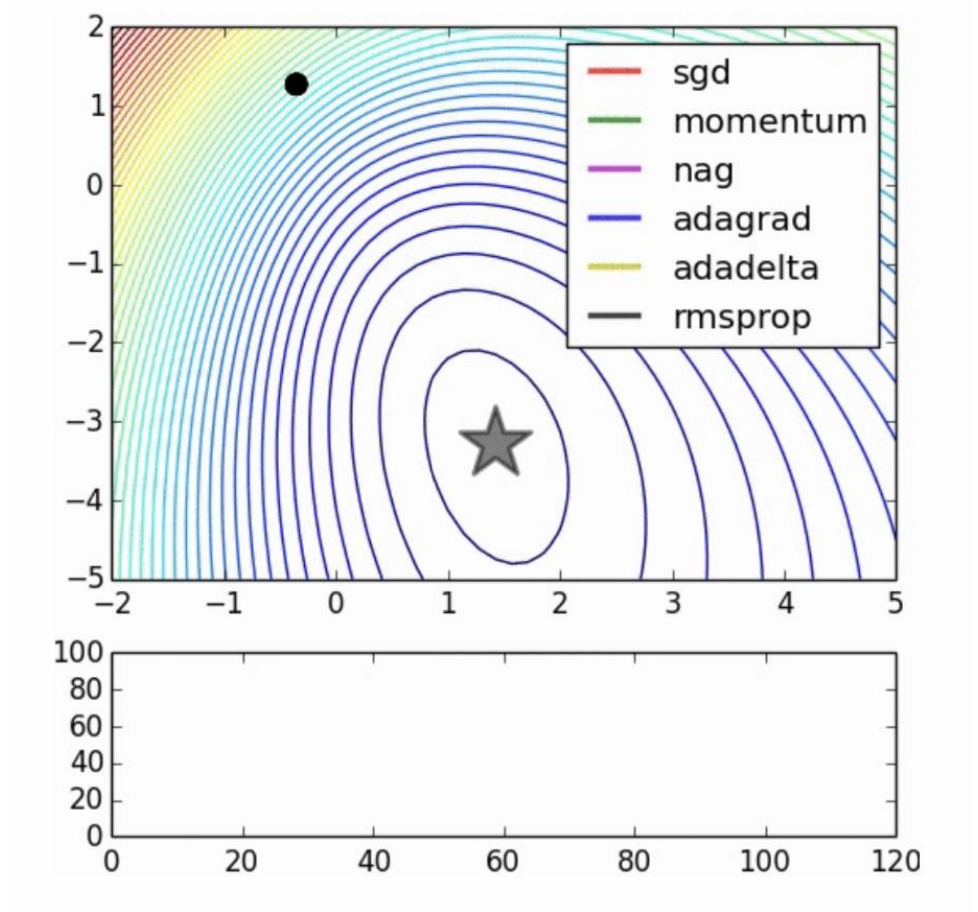
Training a neural network: the main loop

```
while True:
    data_batch = dataset.sample_data_batch()
    loss = network.forward(data_batch)
    dx = network.backward()
    x += -learning_rate * dx
```



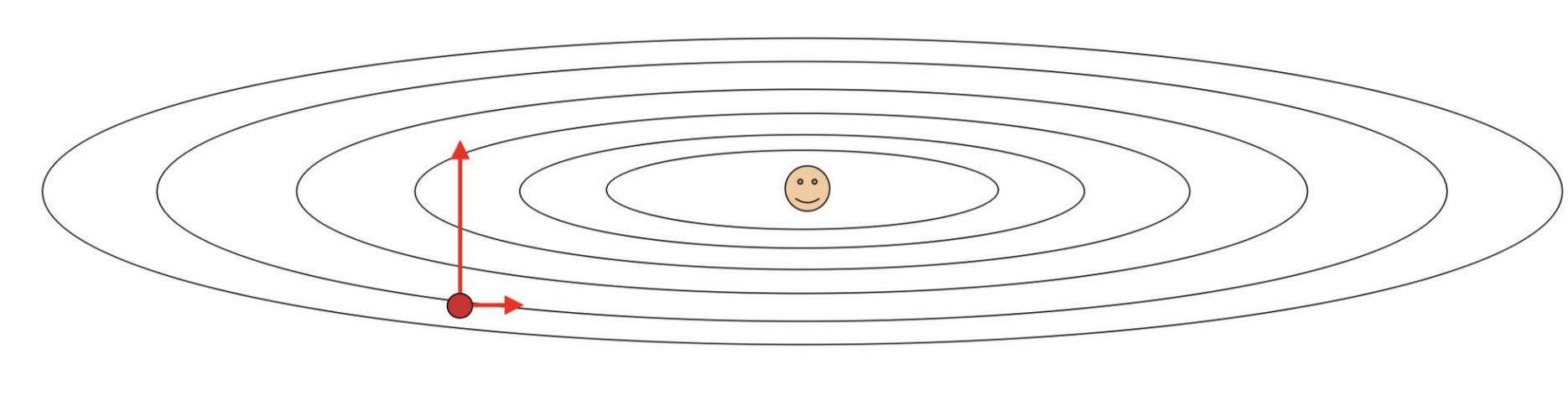
Updating rule: the simple gradient descent

Updating parameters



Updating parameters

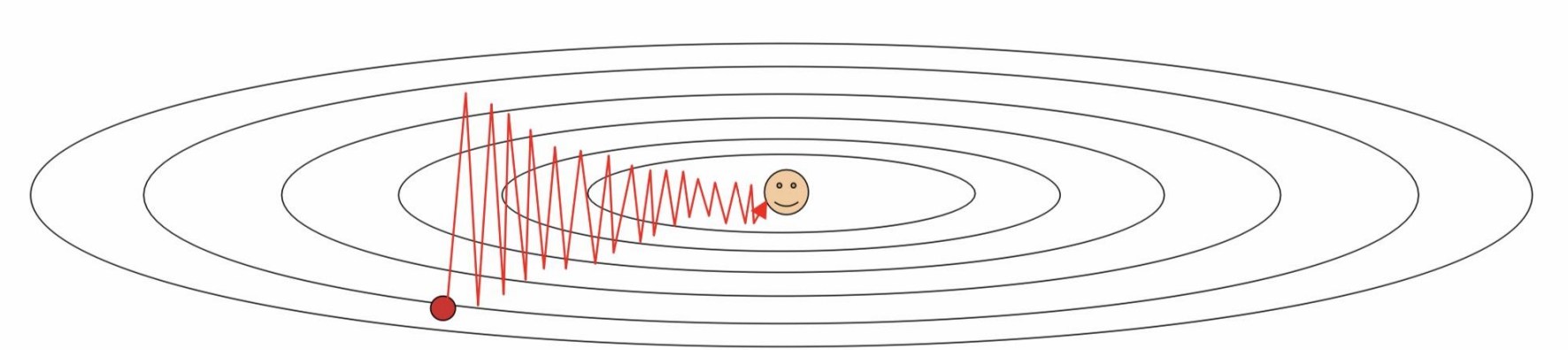
- Cost function: has a steep slope in the vertical direction and a gentle slope in the horizontal direction.



- If using the decreasing gradient, how will the path of convergence towards the minimum point be?

Updating parameters

- Cost function: has a steep slope in the vertical direction and a gentle slope in the horizontal direction.



- In the horizontal direction, the progress will be very slow and in the vertical direction, it will have irregular movements!

Momentum method

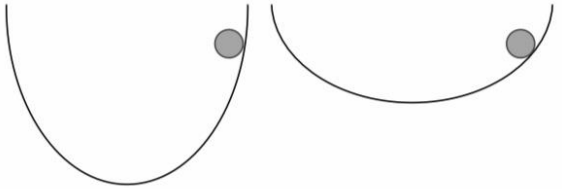
```
# Gradient descent update
x += -learning_rate * dx
```

```
# Momentum update
v = mu * v - learning_rate * dx  # integrate velocity
x += v  # integrate position
```

- Physical interpretation: as a ball moving downwards on the surface of the cost function + friction (mu coefficient)
  - Mu coefficient value: usually 0.5, 0.9 or 0.99

Momentum method

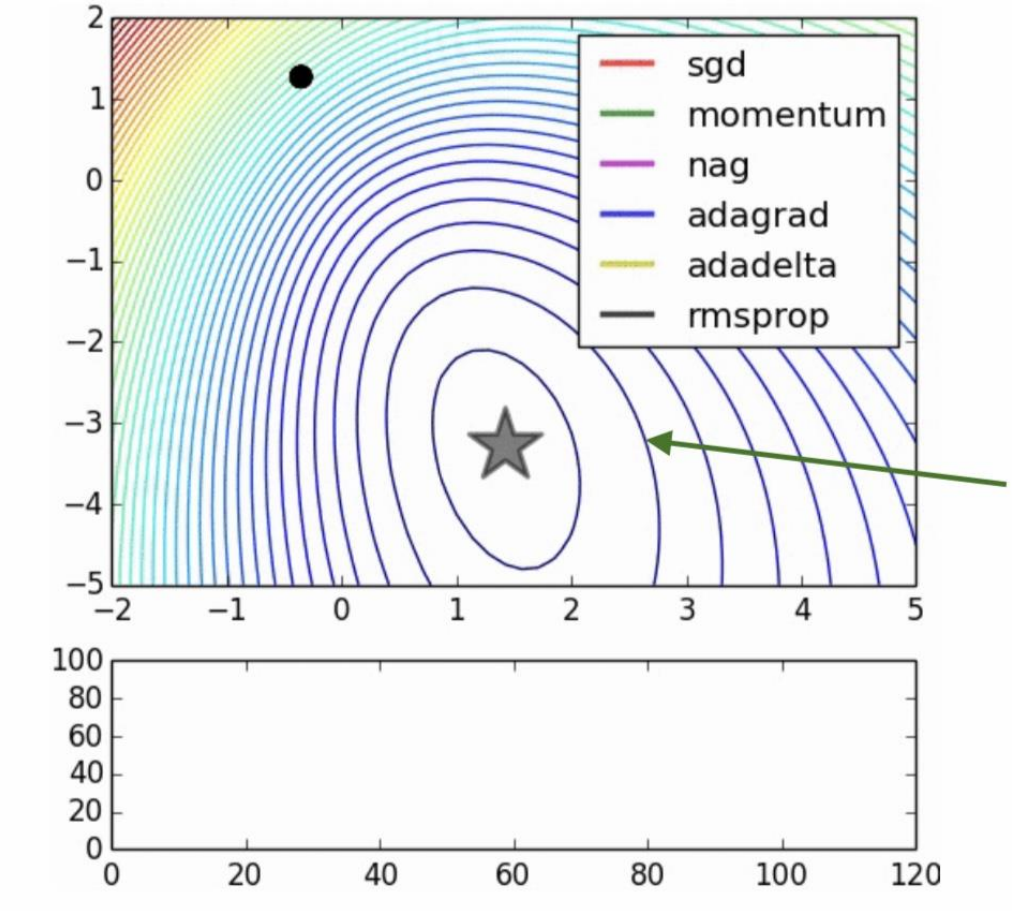
```
# Gradient descent update
x += -learning_rate * dx
```



```
# Momentum update
v = mu * v - learning_rate * dx  # integrate velocity
x += v  # integrate position
```

- In the uphill direction, the speed gradually decreases due to the rapid change of sign.
- In low slope directions, due to the same direction of the velocity vector and the gradient vector, the speed increases gradually.

Momentum method



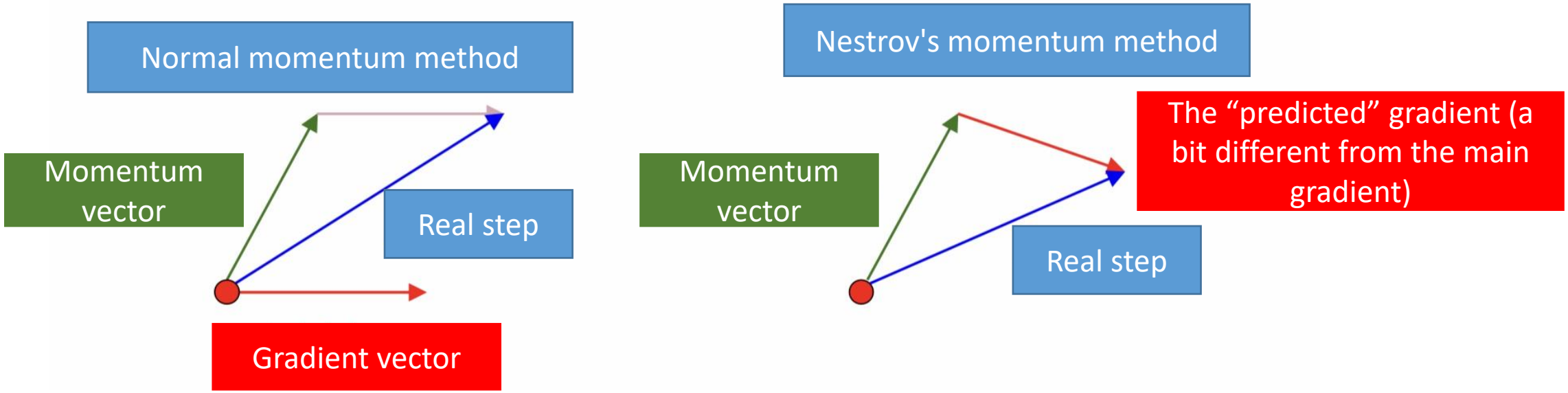
Note that in the momentum method, although we cross the target, this method generally converges faster than the Gradient descent.

Updating parameters: other methods

- First-order methods:
  - Nestrov's momentum method
  - AdaGrad method
  - RMSProp method
  - Adam's method
- Second-order methods:
  - Newton's method
  - Quasi-Newtonian methods (L-BFGS and BFGS)

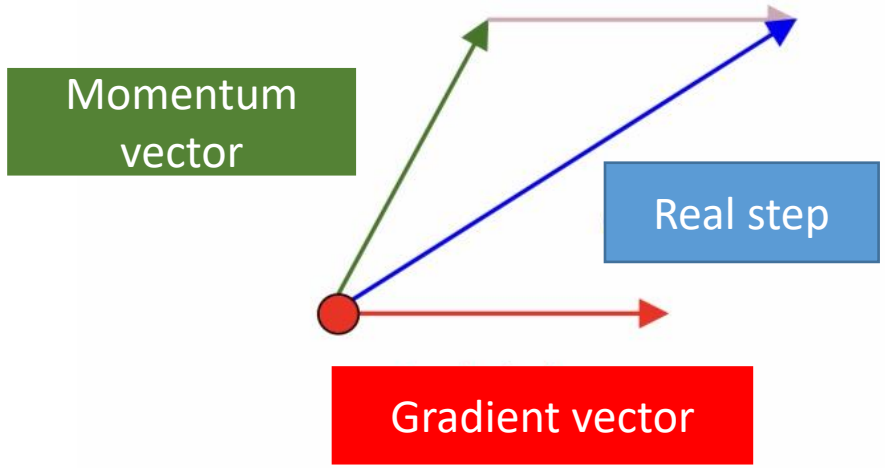
Nestrov's momentum method

```
# Momentum update
v = mu * v - learning_rate * dx  # integrate velocity
x += v  # integrate position
```

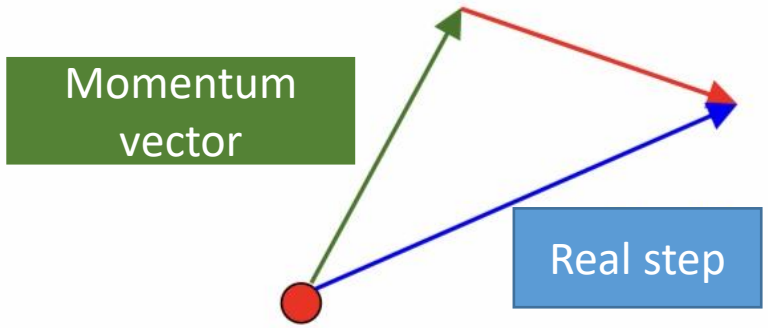


Nestrov's momentum method

Normal momentum method



Nestrov's momentum method



The "predicted" gradient (a bit different from the main gradient)

Nestrov: The only difference...

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Nestrov's momentum method

$$v_t = \mu v_{t-1} - \epsilon \nabla f(\theta_{t-1} + \mu v_{t-1})$$

$$\theta_t = \theta_{t-1} + v_t$$

Relatively inappropriate...
Typically, we have the following vectors:

$$\theta_{t-1}, \nabla f(\theta_{t-1})$$

$$\phi_{t-1} = \theta_{t-1} + \mu v_{t-1}$$

With a variable change and rewrite, the problem is solved.

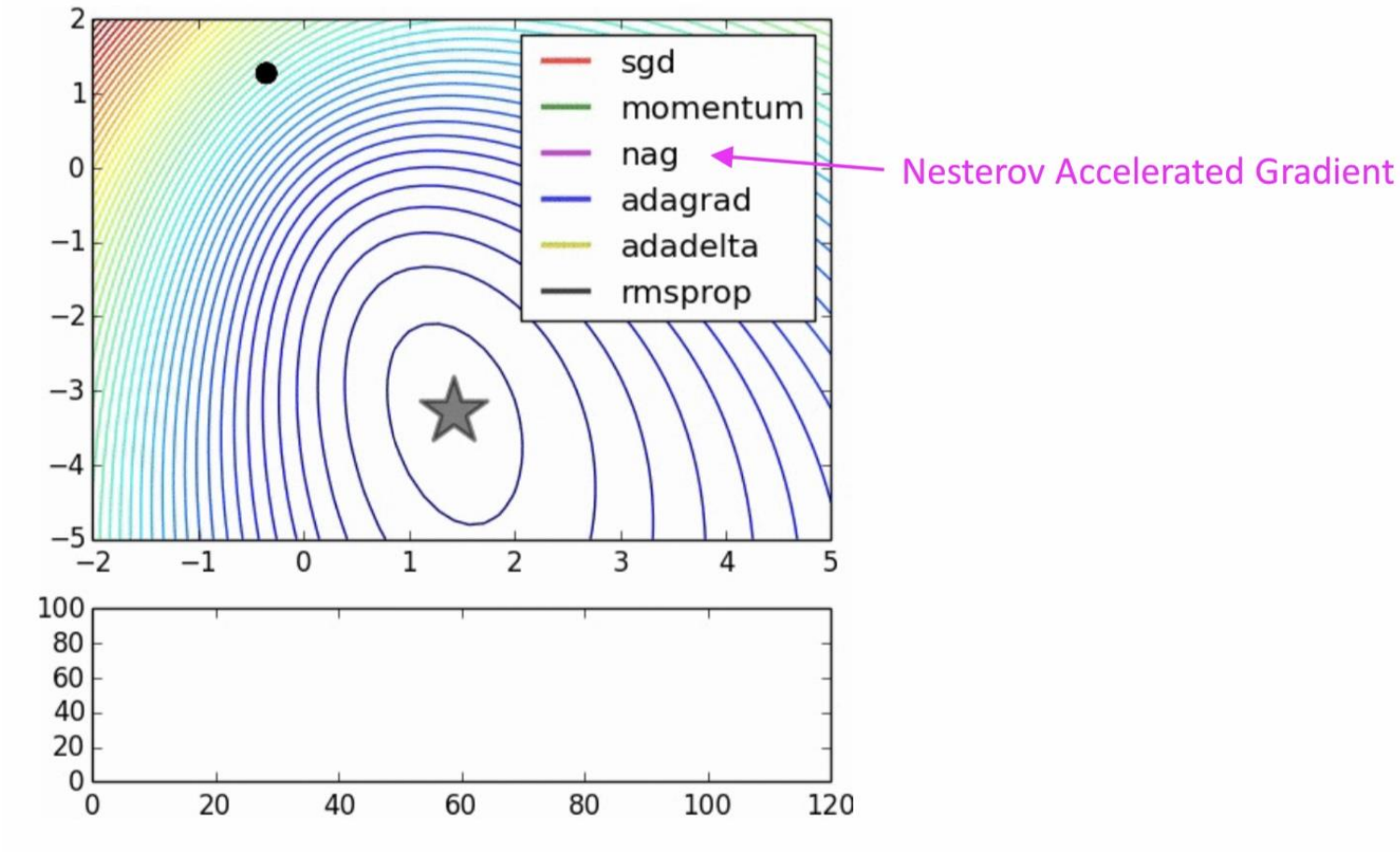
$$v_t = \mu v_{t-1} - \epsilon \nabla f(\phi_{t-1})$$

$$\phi_t = \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t$$

$$\begin{aligned}\phi_t &= \theta_t + \mu v_t = \theta_{t-1} + v_t + \mu v_t \\ &= \phi_{t-1} - \mu v_{t-1} + (1 + \mu) v_t\end{aligned}$$

```
# Nesterov Momentum update rewrite
v_prev = v
v = mu * v - learning_rate * dx
x += -mu * v_prev + (1 + mu) * v
```

Nestrov's momentum method



AdaGrad method

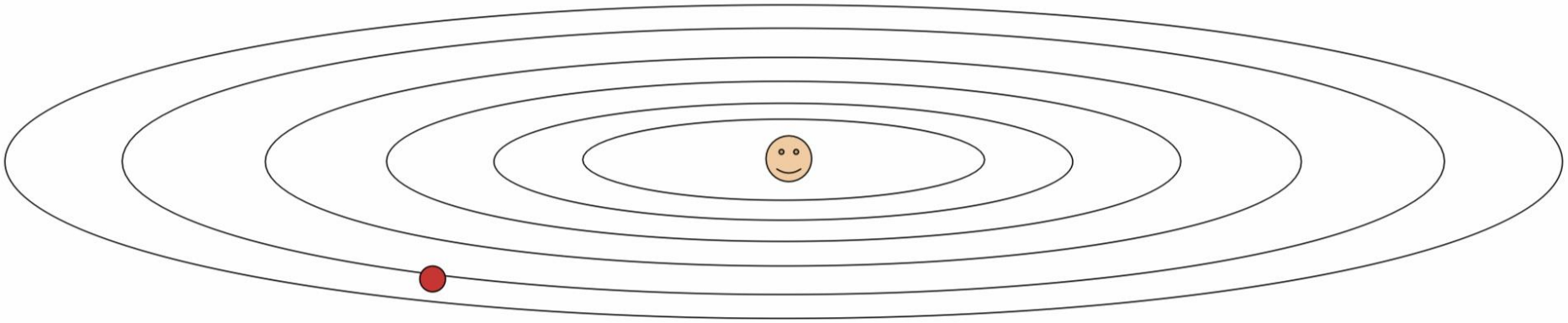
- Scaling of gradients in each dimension, based on the sum of the squares of previous gradient values in the same dimension.

```
# AdaGrad update
cache += dx ** 2
x += -learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

AdaGrad method

- If this method is used, how will the convergence be?

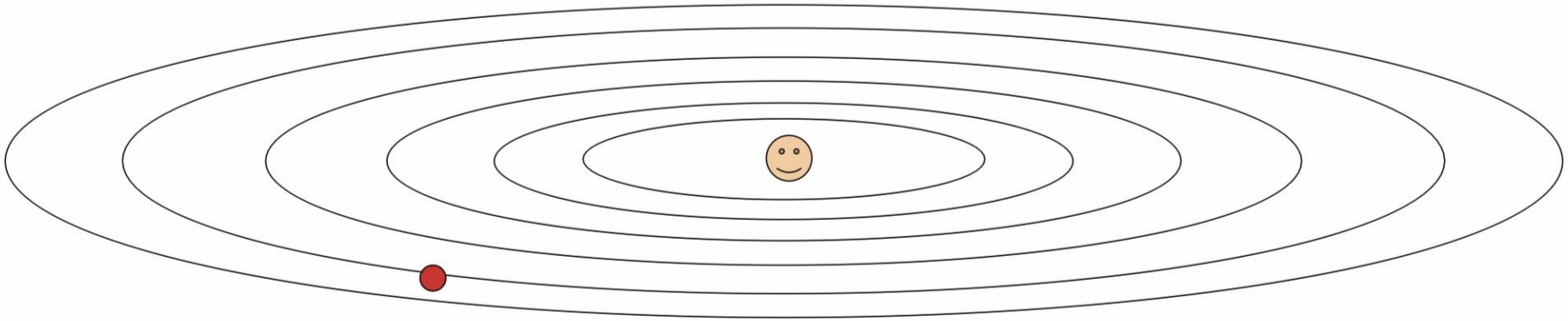
```
# AdaGrad update
cache += dx ** 2
x += -learning_rate * dx / (np.sqrt(cache) + 1e-7)
```



AdaGrad method

- What happens to the learning rate over a long period of time?

```
# AdaGrad update
cache += dx ** 2
x += -learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

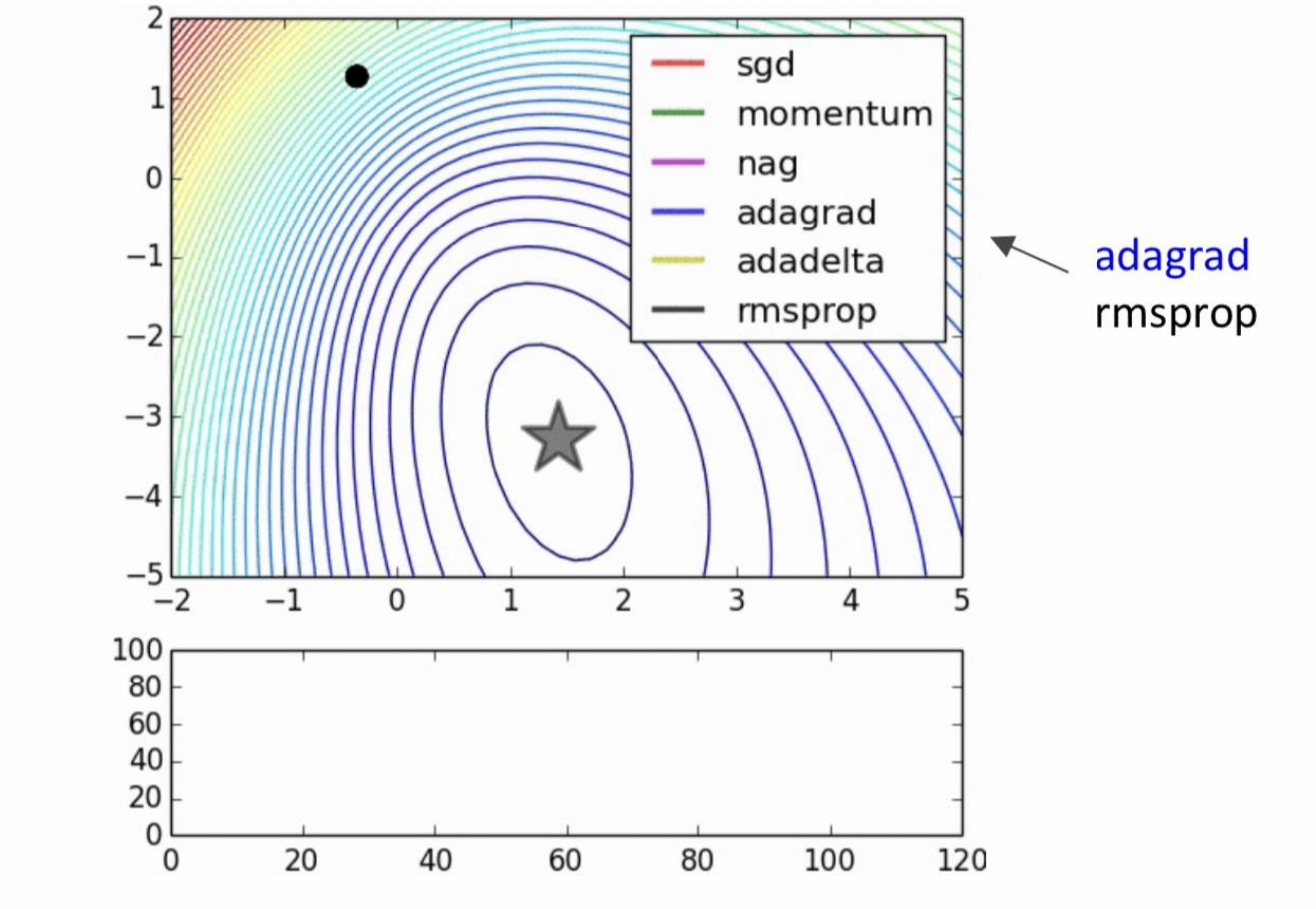


RMSProp Method

```
# AdaGrad update
cache += dx ** 2
x += -learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

```
# RMSProp update
cache = decay_rate * cache + (1 - decay_rate) * dx ** 2
x += -learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

AdaGrad and RMSProp method



Adam Method

```
# Adam update
m = beta1 * m + (1 - beta1) * dx          # first moment
v = beta2 * v + (1 - beta2) * dx ** 2     # second moment
x += -learning_rate * m / (np.sqrt(v) + 1e-7)
```

Somewhat similar to the RMSProp method with momentum

```
# RMSProp update
cache = decay_rate * cache + (1 - decay_rate) * dx ** 2
x += -learning_rate * dx / (np.sqrt(cache) + 1e-7)
```

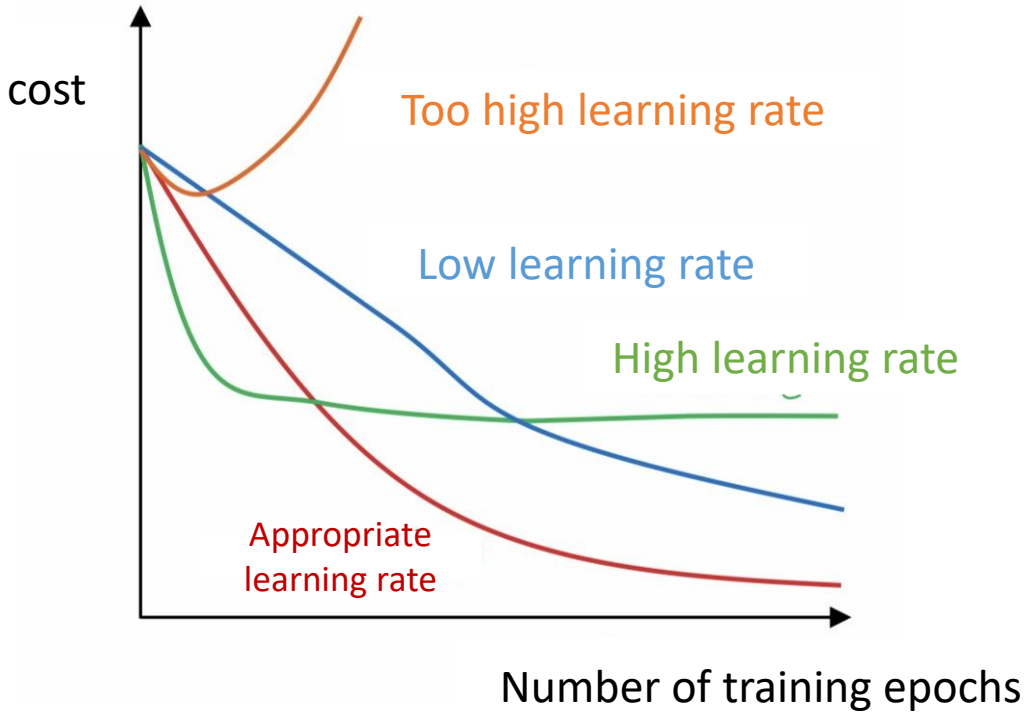
Adam Method

```
# Adam update
m, v =  #... initialize caches to zeros
for t in xrange(1, big_number):
    dx =  #... evaluate gradient
    m  = beta1 * m + (1 - beta1) * dx         # first moment
    v  = beta2 * v + (1 - beta2) * dx ** 2    # second moment
    mb = m / (1 - beta1 ** t)  # correct bias
    vb = v / (1 - beta2 ** t)  # correct bias
    x += -learning_rate * mb / (np.sqrt(vb) + 1e-7)
```

- The reason for correcting the biases is that m and v are zero at the beginning and it takes some time to reach a suitable value.

Learning rate

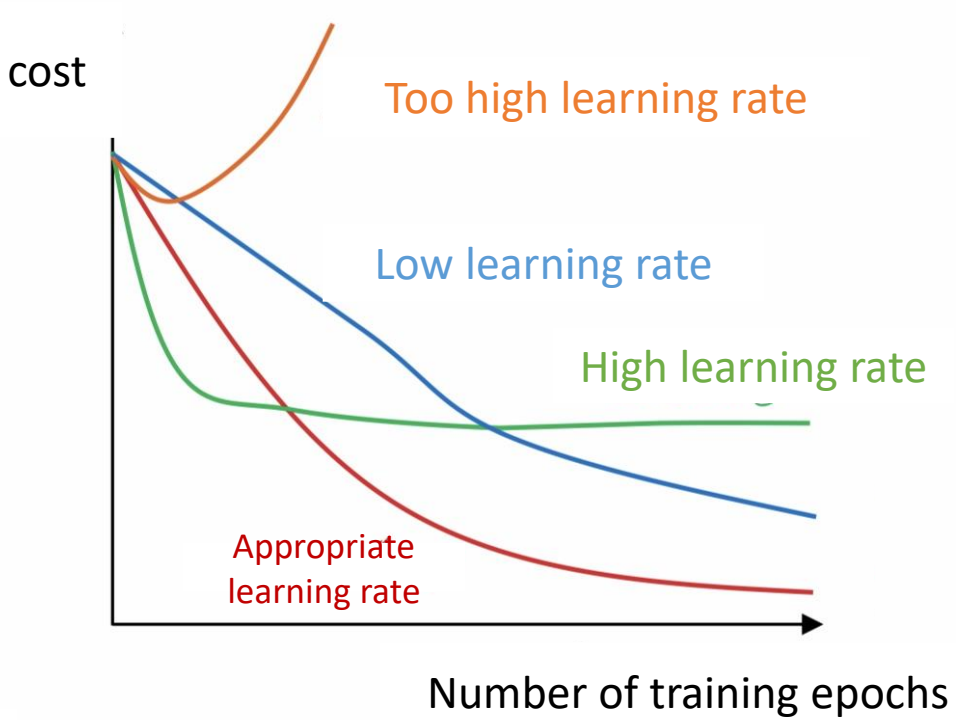
- In all the updating methods described, the value of the learning rate as a hyperparameter must be carefully determined.



Which one is the best value for the learning rate?

Learning rate

- In all the updating methods described, the value of the learning rate as a hyper parameter must be carefully determined.



Decreased learning rate over time!
Step reduction:
For example, after a few training courses, cut the learning rate in half.

Exponential reduction:

$$\alpha = \alpha_0 e^{-kt}$$

decrease proportional to the inverse of t:

$$\alpha = \alpha_0/(1 + kt)$$

Quadratic optimization methods

Second order Taylor expansion:

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

After solving this relationship, to find the critical point, we obtain Newton's updating method:

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

Calculate the differential and set it equal to zero

What is the interesting feature of this update method?

Quadratic optimization methods

Second order Taylor expansion:

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

After solving this relationship, to find the critical point, we obtain Newton's updating method:

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

Note: There are no hyperparameters! (like learning rate)

Why is it not possible to use this method to train deep neural networks in practice?

Quadratic optimization methods

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

- Quasi-Newtonian methods. (BFGS most popular)
  - Instead of computing the inverse of the Hessian matrix, approximate the inverse of the Hessian matrix over time with first-order updates. (Second-order time complexity instead of third-order time complexity)
- L-BFGS method (BFGS with limited memory)
  - Do not save (create) the entire Hessian matrix.

Quadratic optimization methods

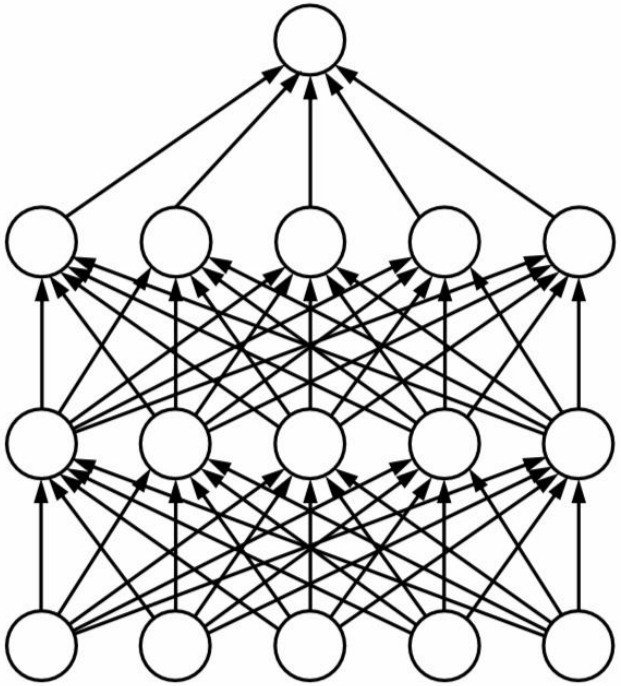
in practice:

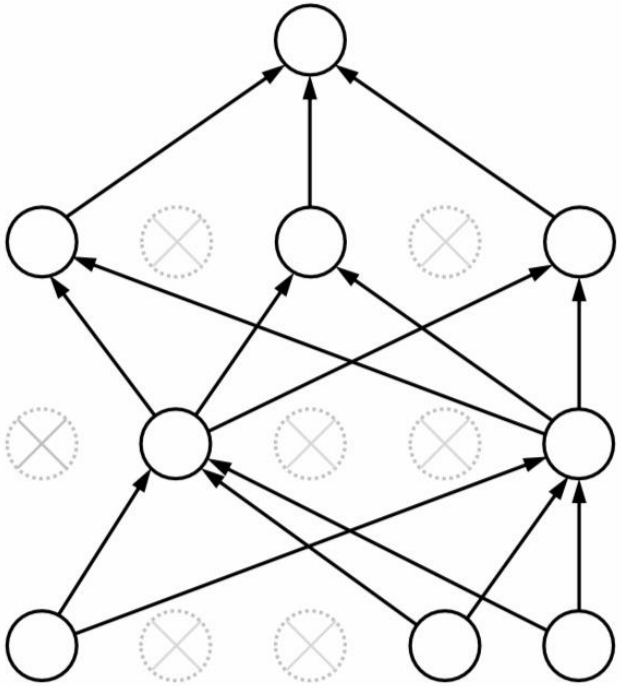
- As the default choice, Adam's method works well in most cases.
- If it is possible for you to update using all the data completely, in each iteration, try the L-BFGS method. (In this case, remember to disable all noise sources.)

Regularization: Random Removal

Regularization: Random Removal (Dropout)

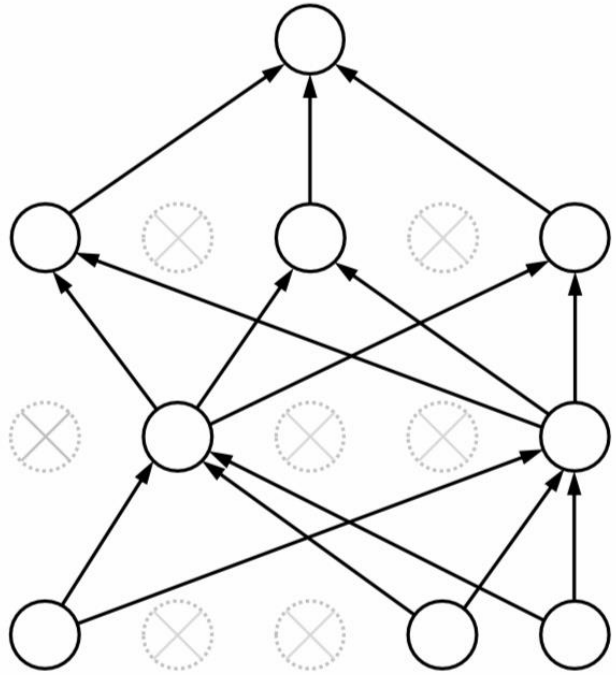
- In forward calculations, randomly set the output of some neurons to zero.





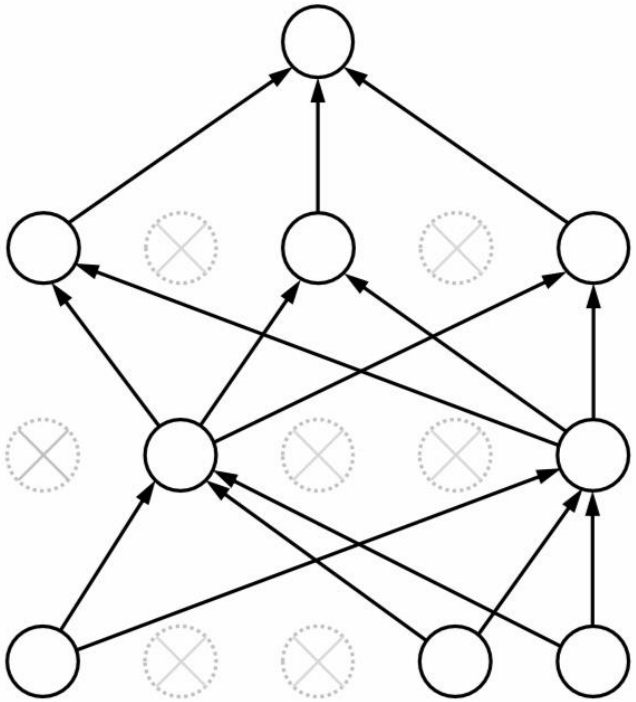
Regularization: Random Removal (Dropout)

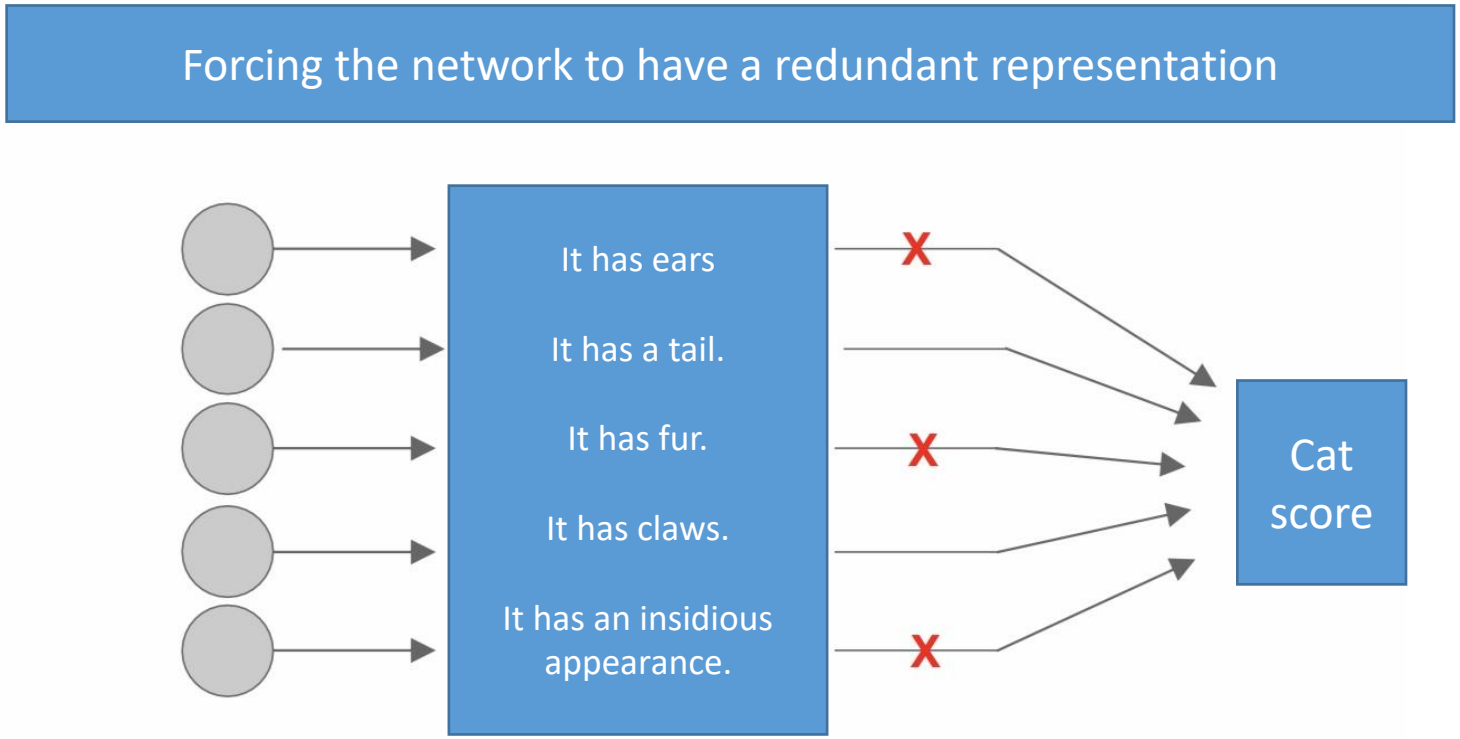
```python
p = 0.5  # probability of keeping a unit active

def train_step(X):
    """ X contains the data """

    # forward pass for a 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p  # first dropout mask
    H1 *= U1  # drop!

    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask
    H2 *= U2  # drop!

    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)
```



Regularization: Random Removal (Dropout)

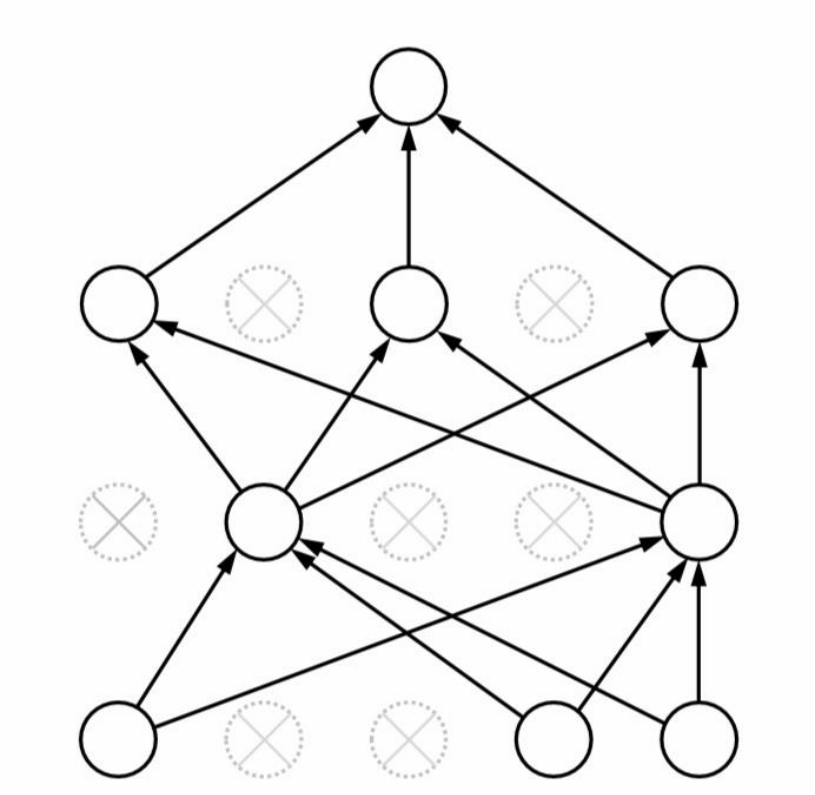
- How could randomly deleting some neurons be a good idea?





Regularization: Random Removal (Dropout)

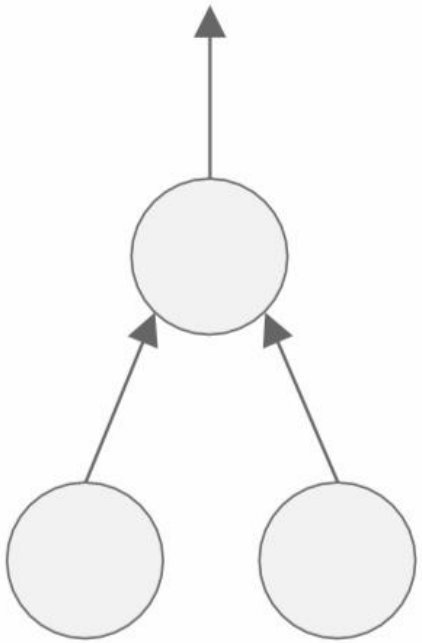
- How could randomly deleting some neurons be a good idea?



Another interpretation:
With random elimination, a large set of models (which have common parameters) are trained.

Dropout in forecast time:

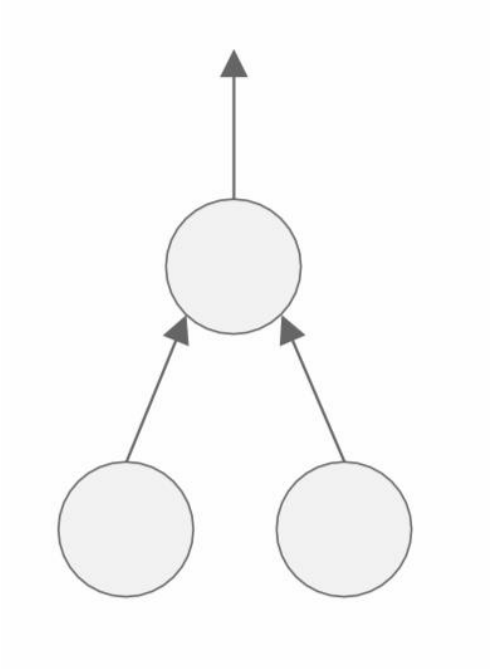
- This can be done with one step of forward calculations! (approximate)
  - By keeping all neurons active (no random deletion)



(It can be shown that this is an approximation of community evaluation of all models.)

Dropout in forecast time:

- This can be done with one step of forward calculations! (approximate)
  - By keeping all neurons active (no random deletion)

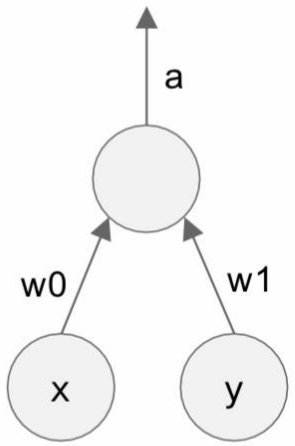


Suppose that the output of this neuron is equal to
a when all the inputs are active during the prediction time.

What is the average output of this neuron during training? (if p = 0.5)

Dropout in forecast time:

- This can be done with one step of forward calculations! (approximate)
  - By keeping all neurons active (no random deletion)



In forecast time:

a = w0*x + w1*y

In training time:

E[a] = ¼ * (w0*0 + w1*0
+ w0*0 + w1*y
+ w0*x + w1*0
+ w0*x + w1*y)

= ¼ * (2 w0*x + 2 w1*y)

= ½ * **(w0*x + w1*y)**

If p = 0.5, the output during prediction will be 2 times the output during training. To compensate for this issue, the amount of activity of this neuron should be multiplied by a factor of 1.2 during the prediction time.

Dropout in forecast time:

```
def predict(X):
    # ensemble forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

- During anticipation, all neurons are always active:
- Therefore, we must change the activity values of neurons in such a way that:
- Output at prediction time = average output at training time

Dropout

```python
p = 0.5  # probability of keeping a unit active

def train_step(X):
    """ X contains the data """

    # forward pass for a 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = np.random.rand(*H1.shape) < p  # first dropout mask!
    H1 *= U1  # drop!

    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = np.random.rand(*H2.shape) < p # second dropout mask!
    H2 *= U2  # drop!

    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensemble forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1) * p  # NOTE: scale the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2) * p  # NOTE: scale the activations
    out = np.dot(W3, H2) + b3
```

Random removal in training time

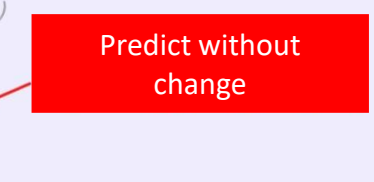
scaling in prediction time

This implementation is not recommended.

Dropout

Common implementation

```
p = 0.5  # probability of keeping a unit active

def train_step(X):
    """ X contains the data """

    # forward pass for a 3-layer neural network
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p  # first dropout mask! Notice /p
    H1 *= U1  # drop!

    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask! Notice /p
    H2 *= U2  # drop!

    out = np.dot(W3, H2) + b3

    # backward pass: compute gradients... (not shown)
    # perform parameter update... (not shown)

def predict(X):
    # ensemble forward pass
    H1 = np.maximum(0, np.dot(W1, X) + b1)  # NOTE: no scaling necessary
    H2 = np.maximum(0, np.dot(W2, H1) + b2)  # NOTE: no scaling necessary
    out = np.dot(W3, H2) + b3
```

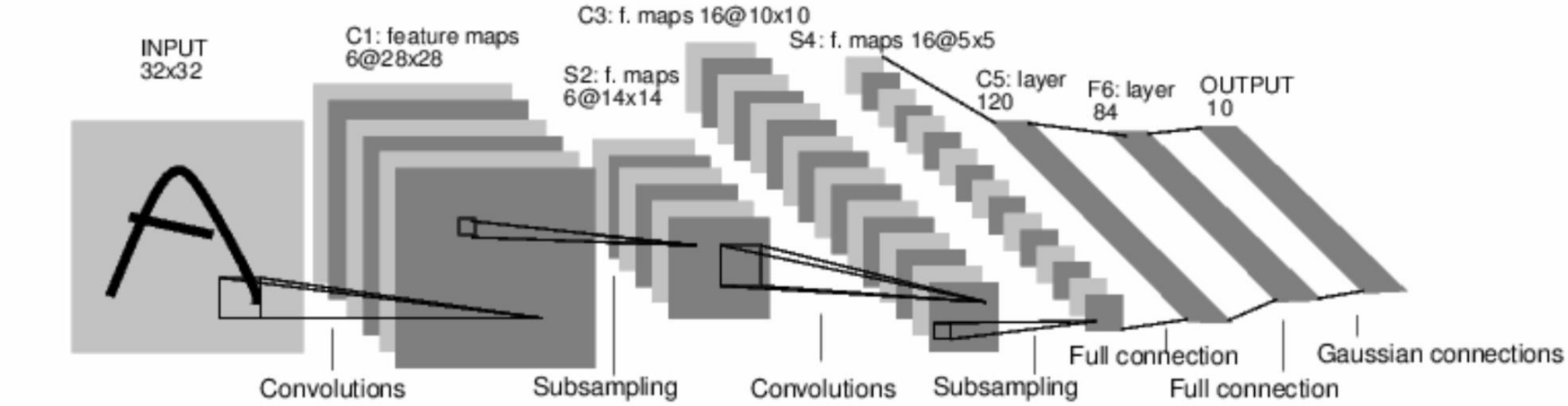


Deep Learning – Convolutional Neural Networks

Why deep learning?

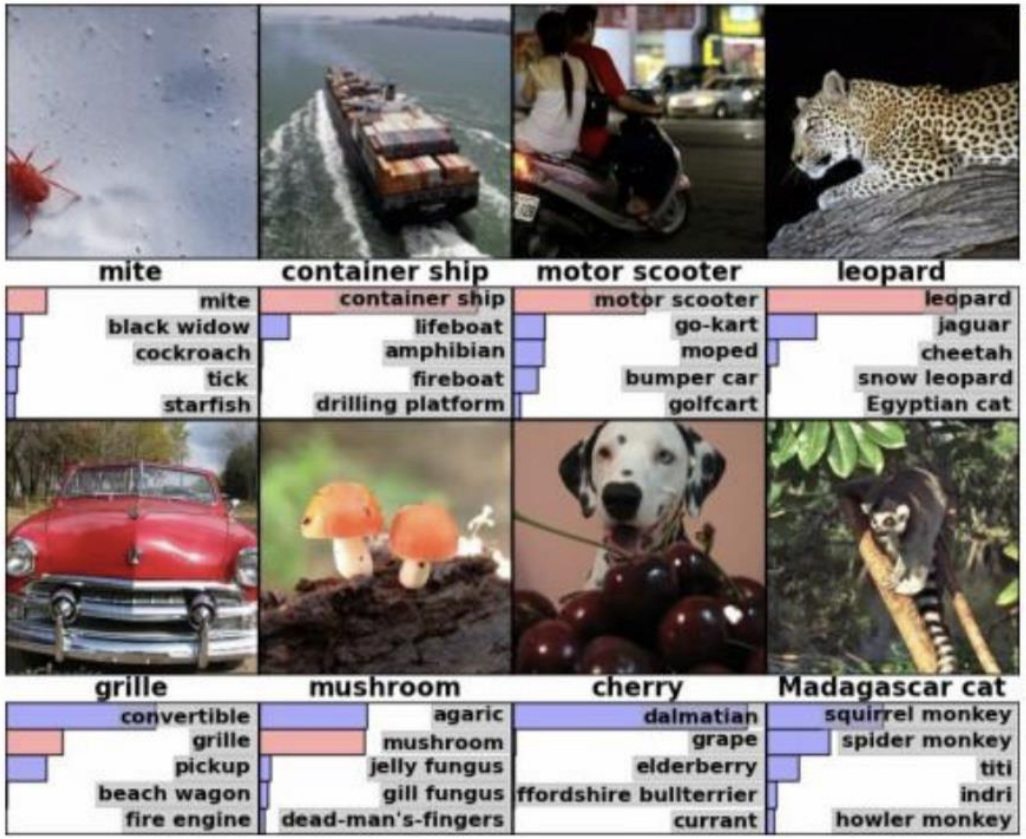
- Learning more complex functions, by combining a sequence of simpler functions in each layer.
  - More layers mean more learning capacity and more complex problem-solving.
  - Learning at different levels of abstraction.
- Using all available data for training purposes.
  - Avoid overfitting!
- Use cheap parallel computing.
  - Graphics processors (GPU)

Convolutional Neural Networks



Convolutional Neural Networks

classification

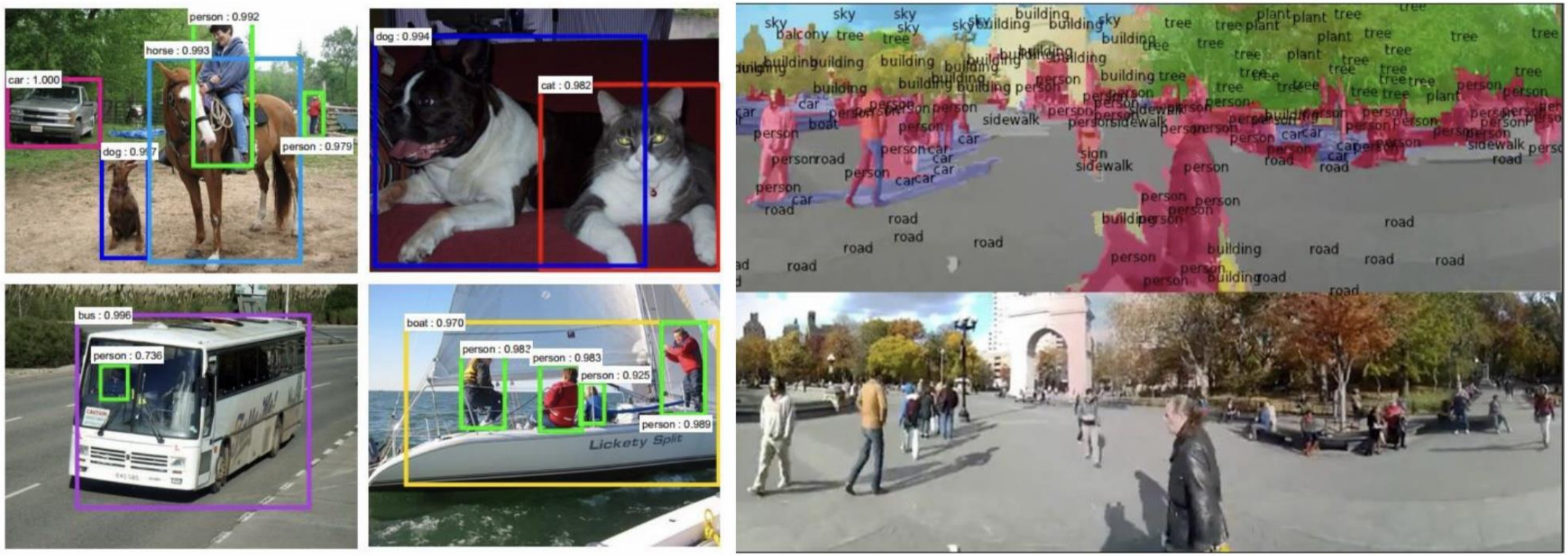


recovery

Convolutional Neural Networks

Identification

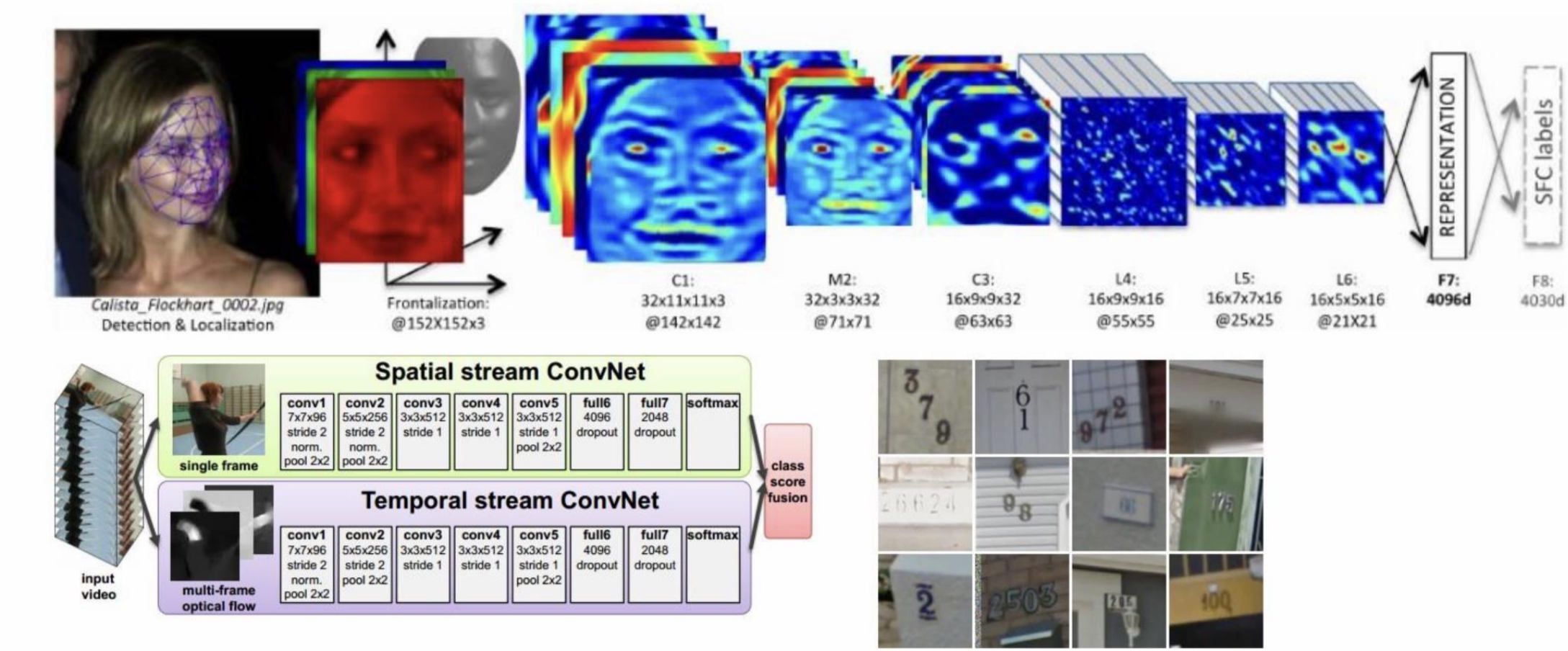
segmentation



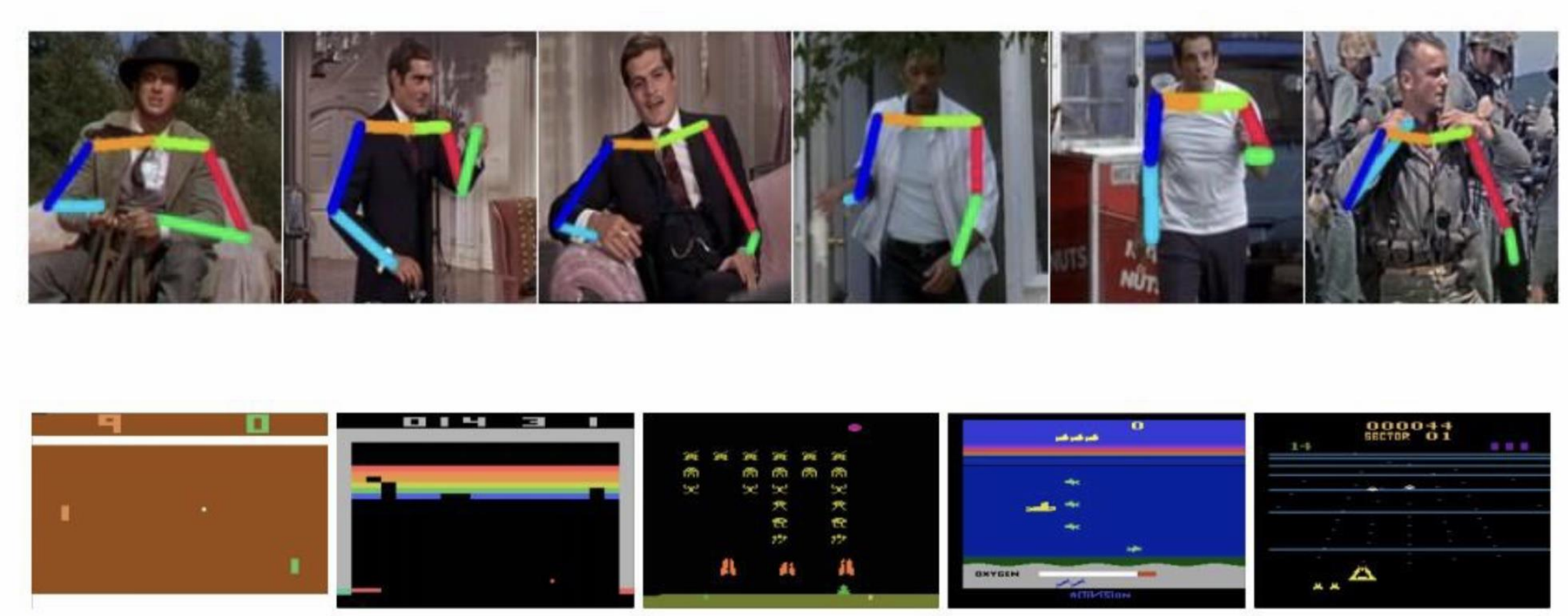
Convolutional Neural Networks

Driverless car

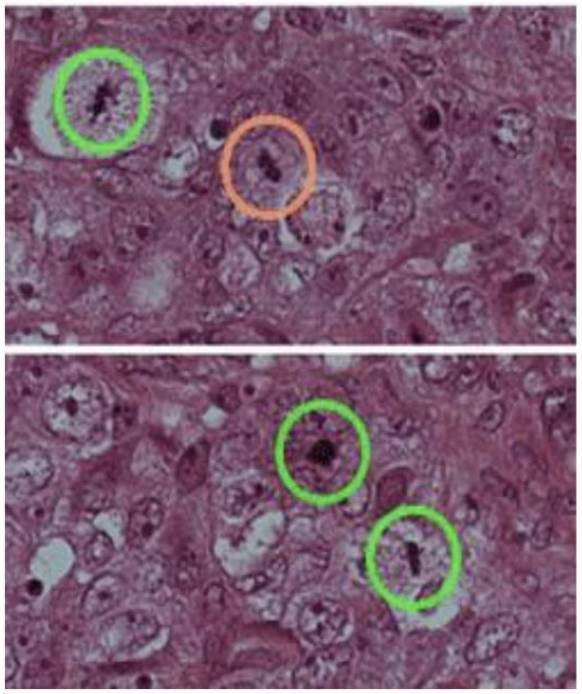
Convolutional Neural Networks



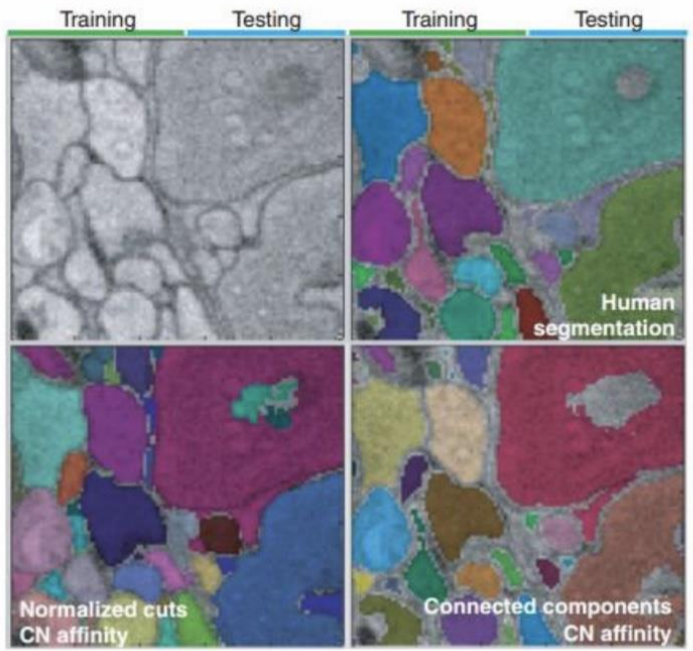
Convolutional Neural Networks

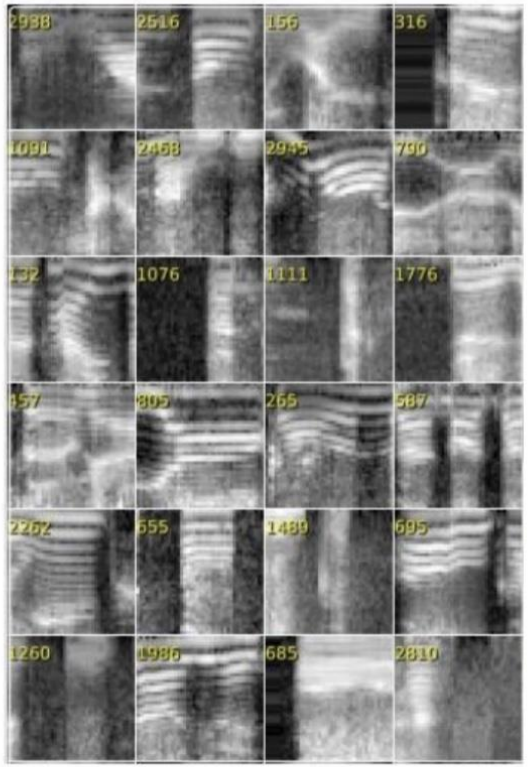


Convolutional Neural Networks



Convolutional Neural Networks



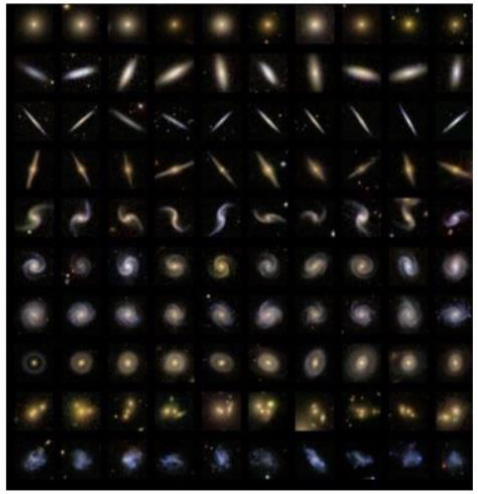


I caught this movie on the Sci-Fi channel recently. It actually turned out to be pretty decent as far as B-list horror/suspense films go. **Two guys (one naive and one loud mouthed a **) take a road trip to stop a wedding but have the worst possible luck when a maniac in a freaky, make-shift tank/truck hybrid decides to play cat-and-mouse with them.** Things are further complicated when they pick up a ridiculously whorish hitchhiker. What makes this film unique is that the combination of comedy and terror actually work in this movie, unlike so many others. The two guys are likable enough and there are some good chase/suspense scenes. Nice pacing and comic timing make this movie more than passable for the horror/slasher buff. **Definitely worth checking out.**

I just saw this on a local independent station in the New York City area. **The cast showed promise but when I saw the director, George Cosmotos, I became suspicious. And sure enough, it was every bit as bad, every bit as pointless and stupid as every George Cosmotos movie I ever saw.** He's like a stupid man's Michael Bey – with all the awfulness that accolade promises. There's no point to the conspiracy, no burning issues that urge the conspirators on. We are left to ourselves to connect the dots from one bit of graffiti on various walls in the film to the next. Thus, the current budget crisis, the war in Iraq, Islamic extremism, the fate of social security, 47 million Americans without health care, stagnating wages, and the death of the middle class are all subsumed by the sheer terror of graffiti. A truly, stunningly idiotic film.

Graphics is far from the best part of the game. **This is the number one best TH game in the series.** Next to Underground. **It deserves strong love. It is an insane game.** There are massive levels, massive unlockable characters... it's just a massive game. **Waste your money on this game. This is the kind of money that is wasted properly.** And even though graphics suck, thats doesn't make a game good. Actually, the graphics were good at the time. Today the graphics are crap. WHO CARES? As they say in Canada, This is the fun game, aye. (You get to go to Canada in THPS3) Well, I don't know if they say that, but they might. who knows. Well, Canadian people do. Wait a minute, I'm getting off topic. This game rocks. Buy it, play it, enjoy it, love it. It's PURE BRILLIANCE.

The first was good and original. I was a not bad horror/comedy movie. So I heard a second one was made and I had to watch it . What really makes this movie work is Judd Nelson's character and the sometimes clever script. **A pretty good script for a person who wrote the Final Destination films and the direction was okay.** Sometimes there's scenes where it looks like it was filmed using a home video camera with a grainy - look. Great made - for - TV movie. **It was worth the rental and probably worth buying just to get that nice eerie feeling and watch Judd Nelson's Stanley doing what he does best.** I suggest newcomers to watch the first one before watching the sequel, just so you'll have an idea what Stanley is like and get a little history background.

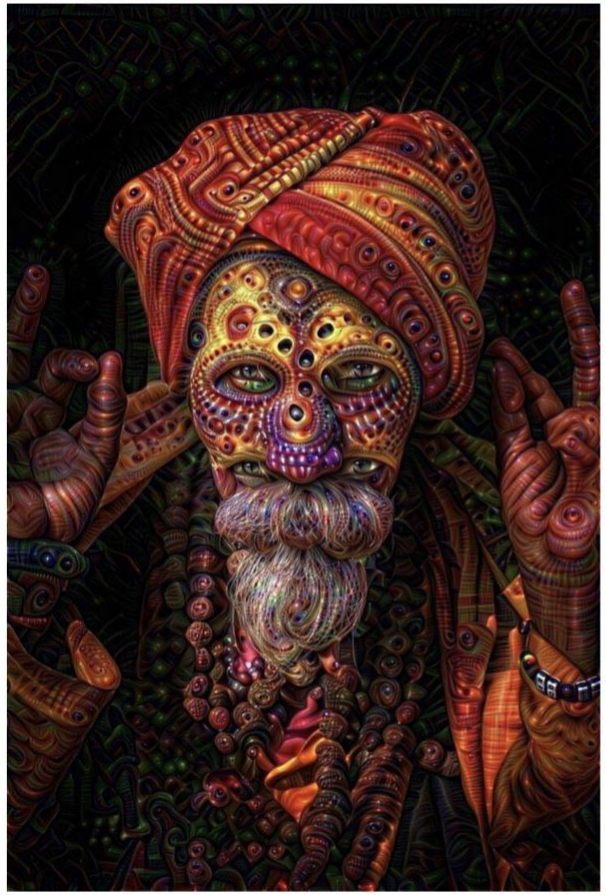


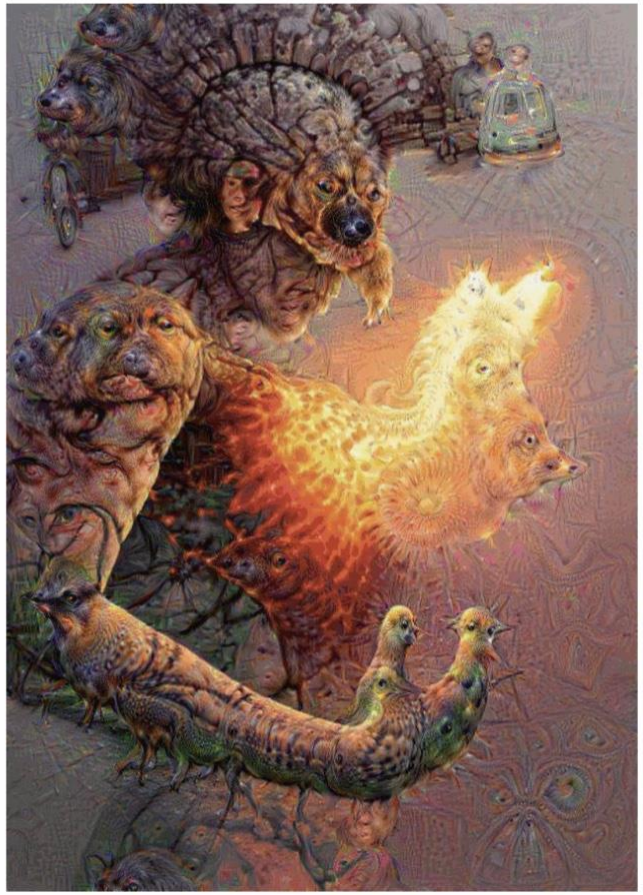
Convolutional Neural Networks

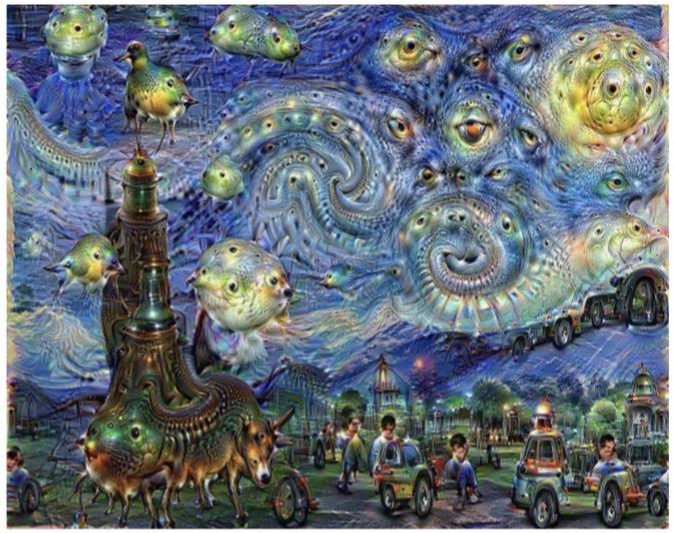
Convolutional Neural Networks

Image Captioning

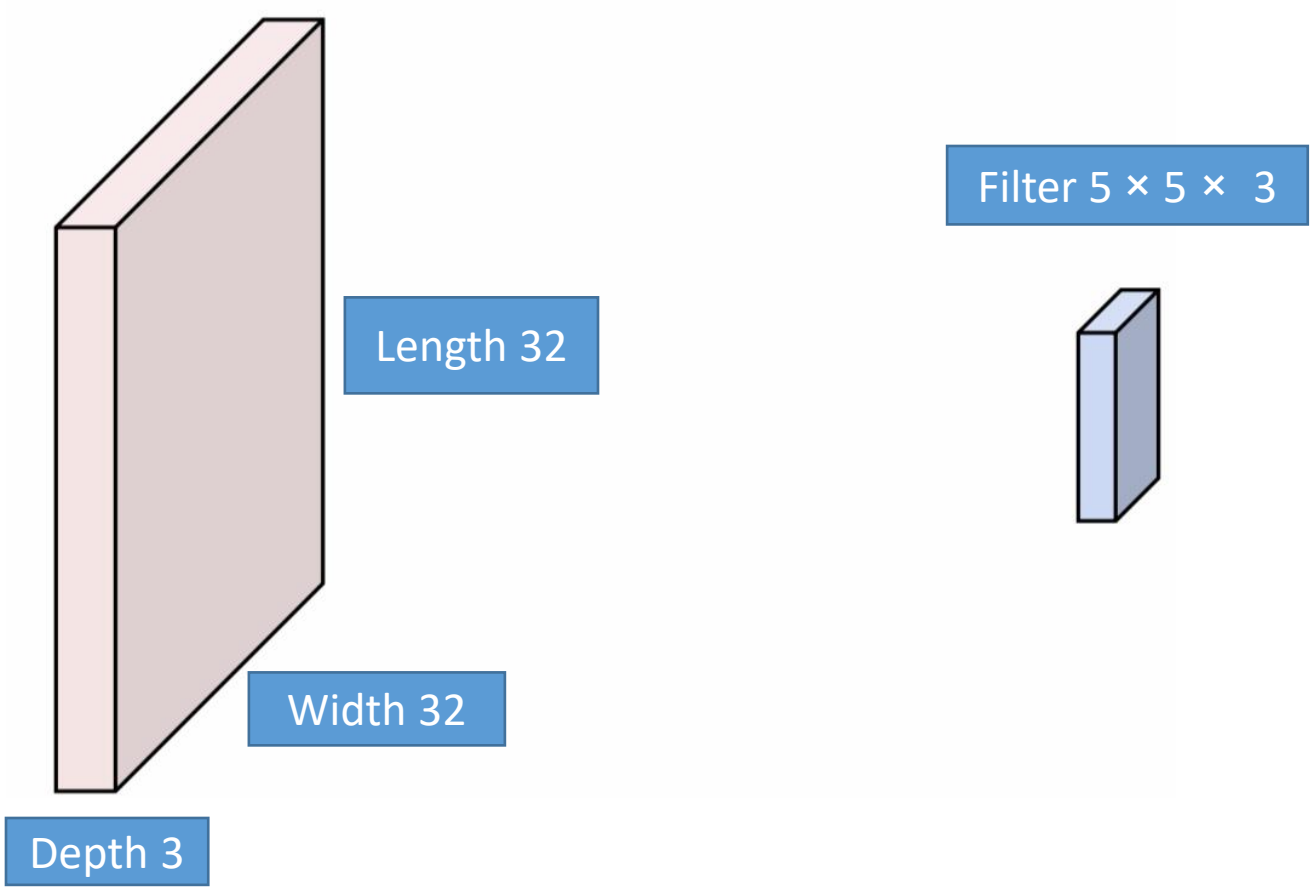
Convolutional Neural Networks







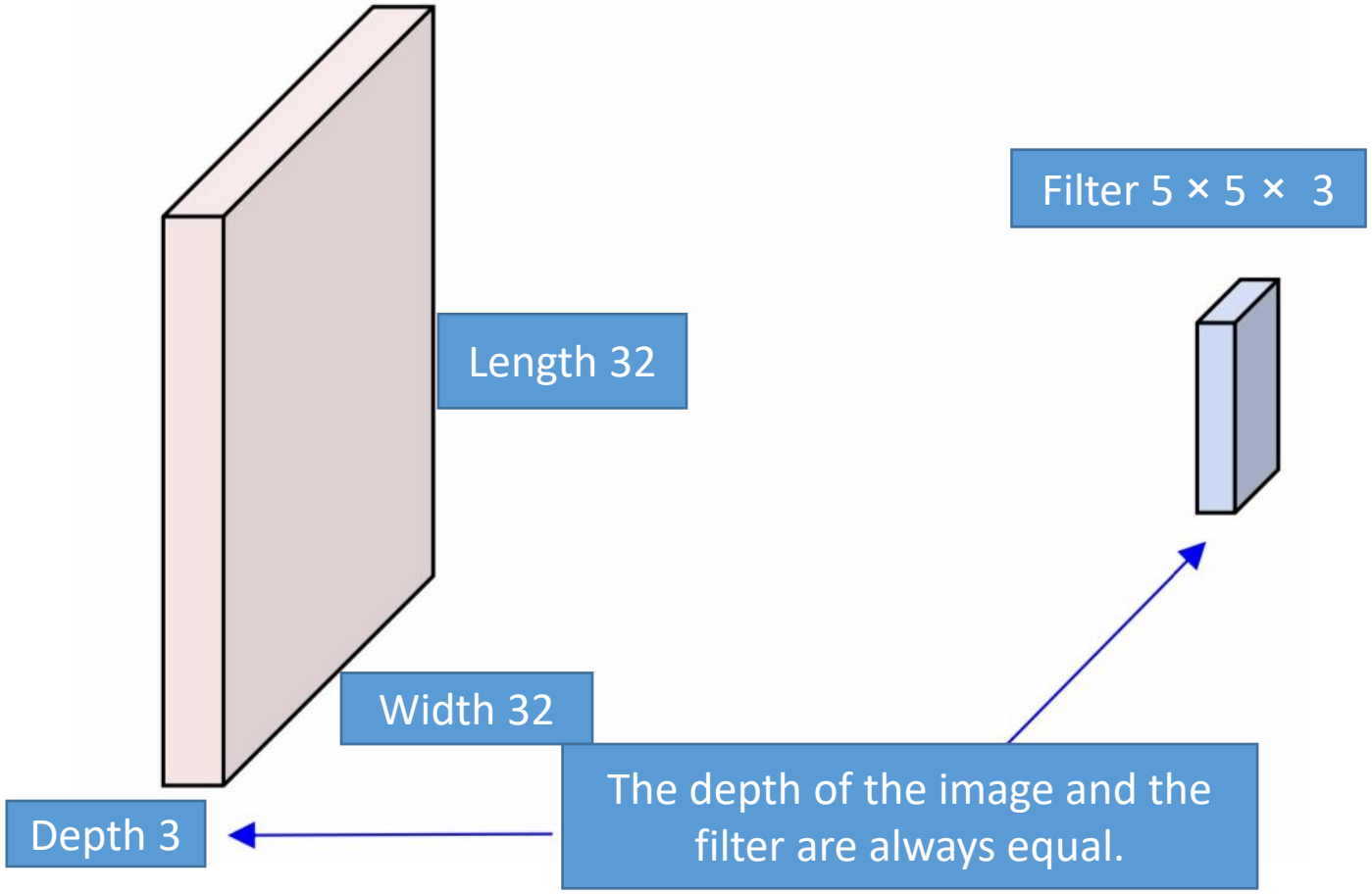
Convolution Layer



Convolution filter with image

That is, "Move the filter on the image and calculate the dot product (internal multiplication) of the filter and the part of the image on which the filter is located each time.

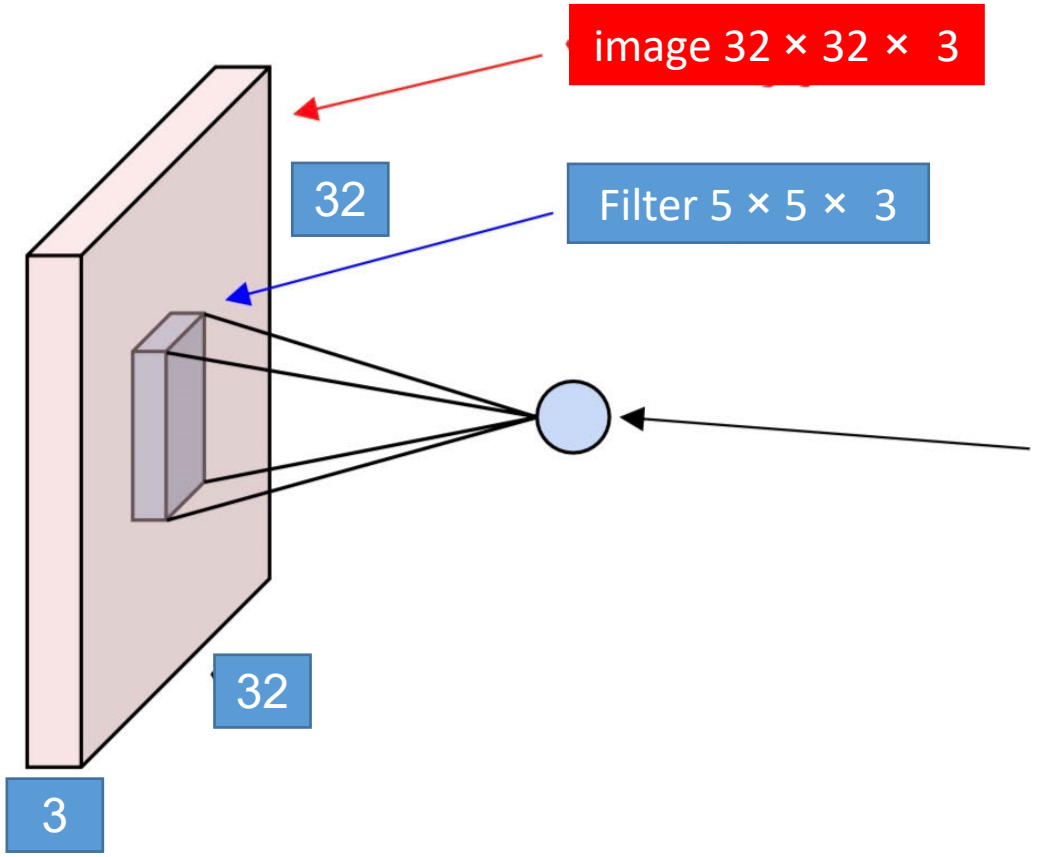
Convolution Layer



Convolution filter with image

That is, "Move the filter on the image and calculate the dot product (internal multiplication) of the filter and the part of the image on which the filter is located each time.

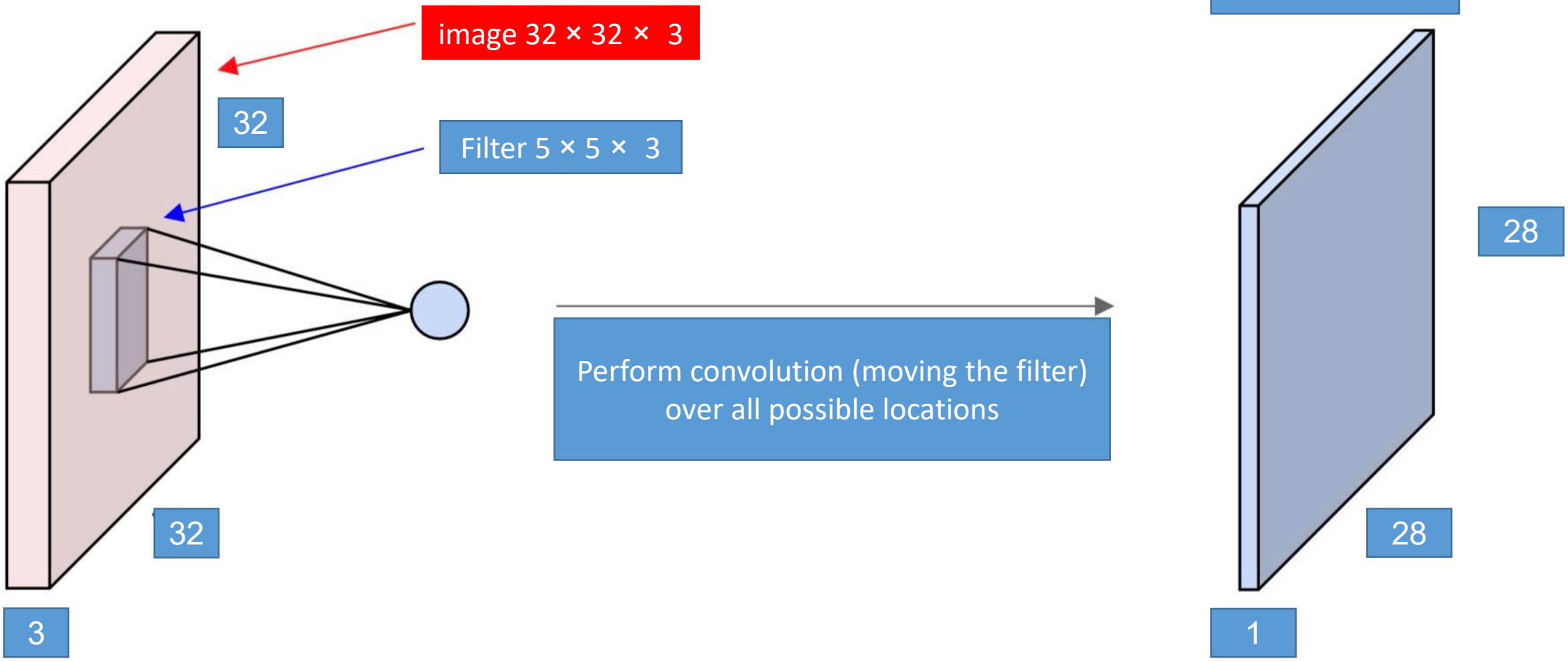
Convolution Layer



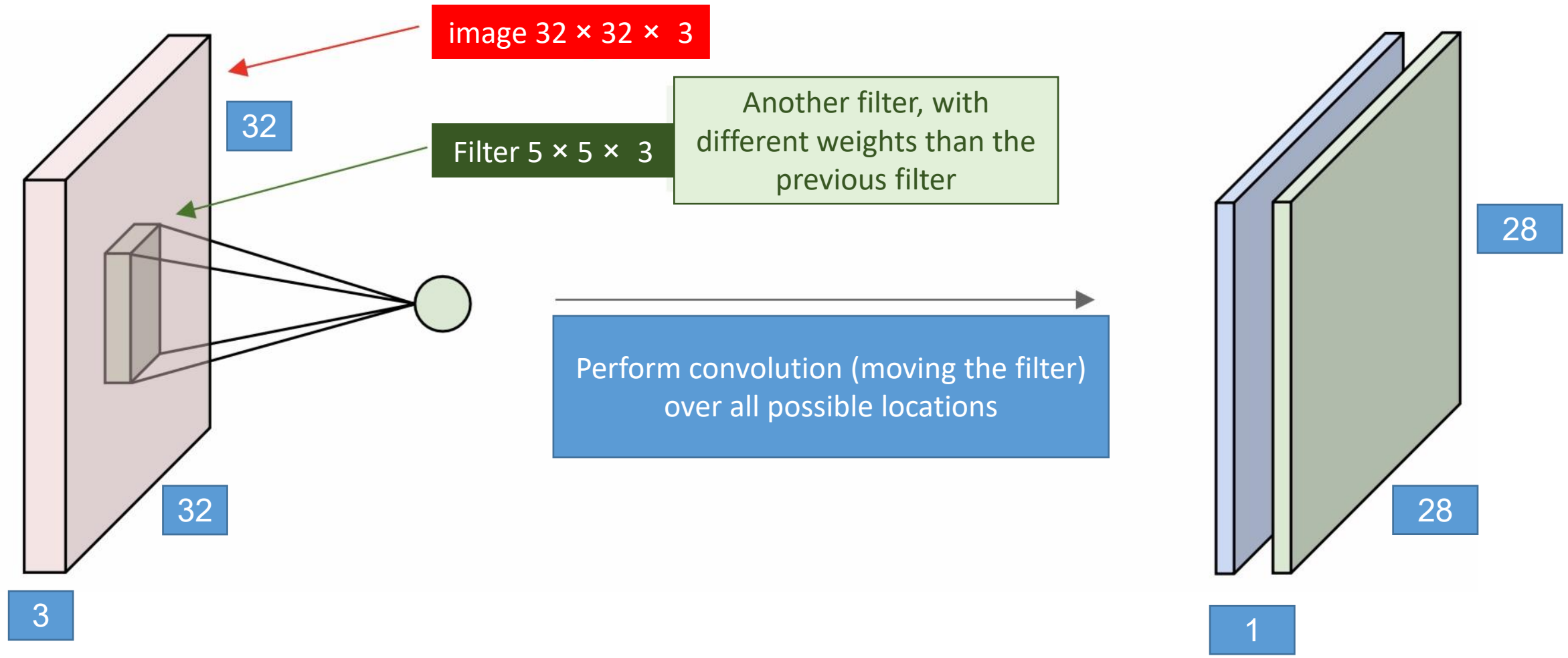
A number:
Obtained from the inner multiplication of the filter and a small part of the image on which the filter is located.
(that is, the inner product of two 75-dimensional vectors + bias)

$$w^T x + b$$

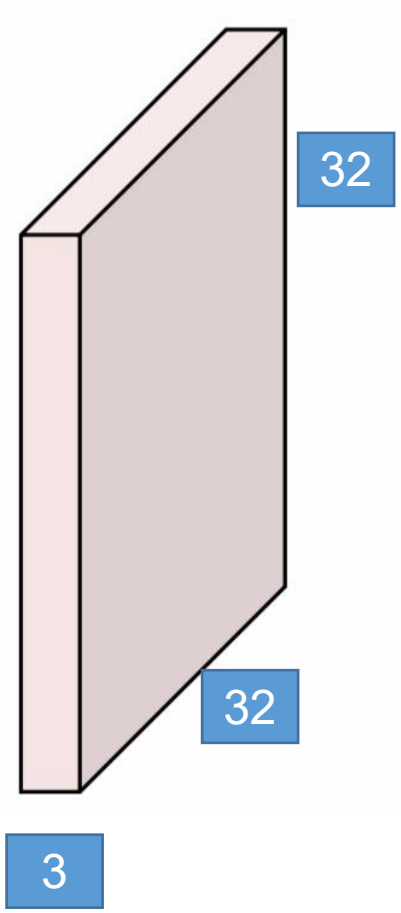
Convolution Layer



Convolution Layer



Convolution Layer

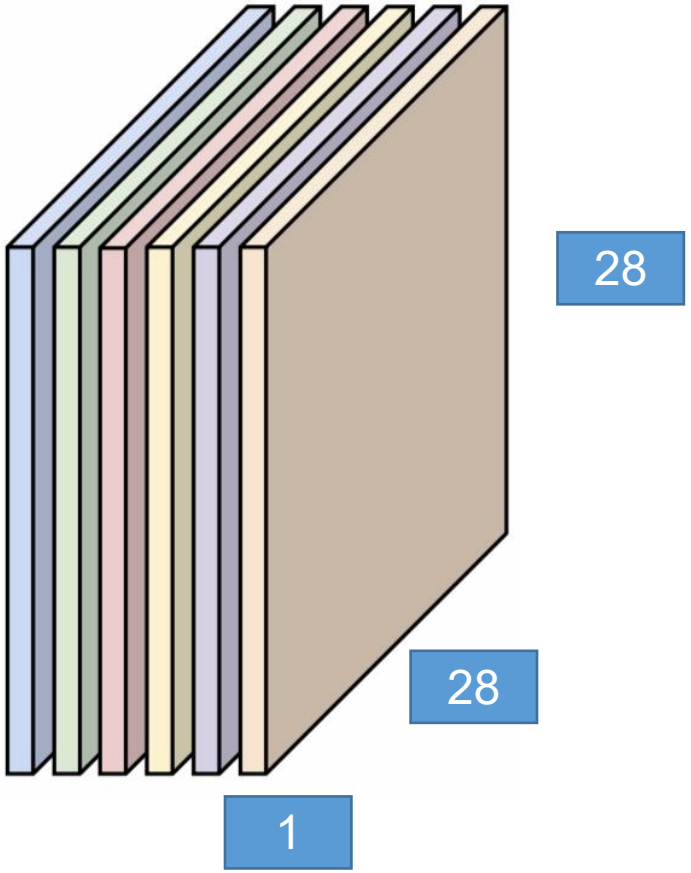


Using 6 filters with dimensions of 5 x 5, 6 activity maps are obtained.

Convolution Layer

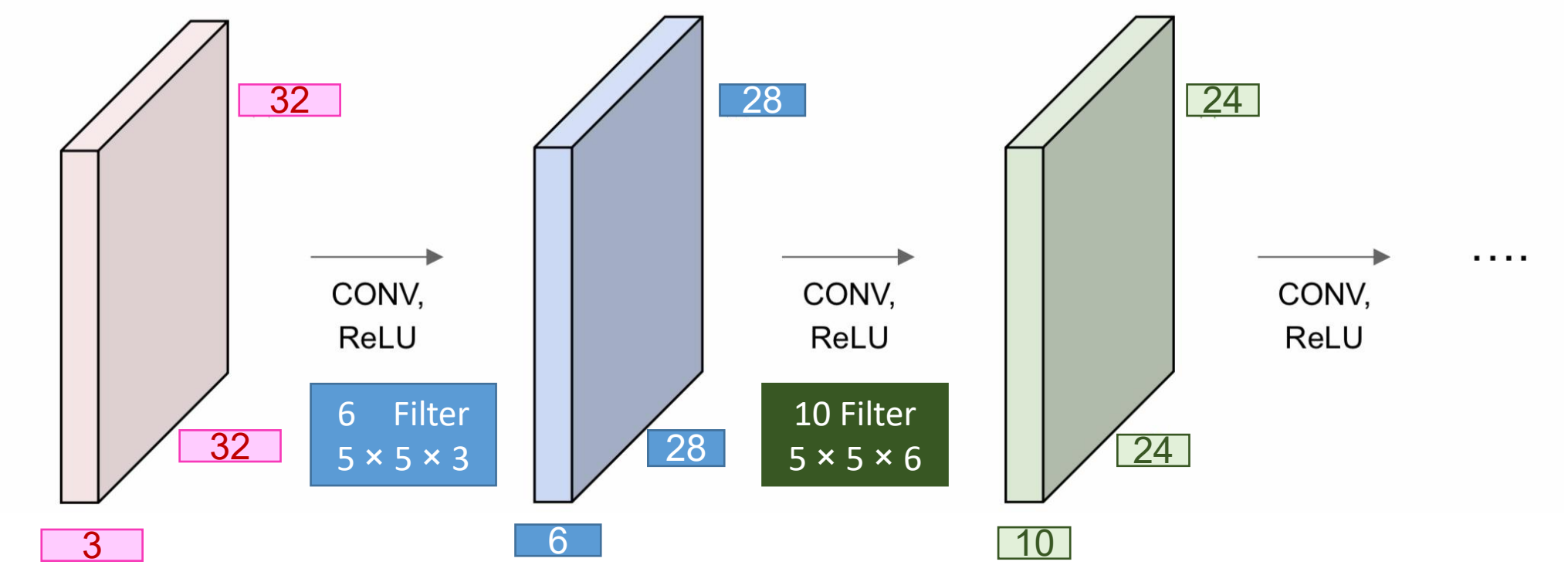
By superimposing these activity maps, a "new image" of dimensions 28 x 28 x 6 is obtained.

Activation maps

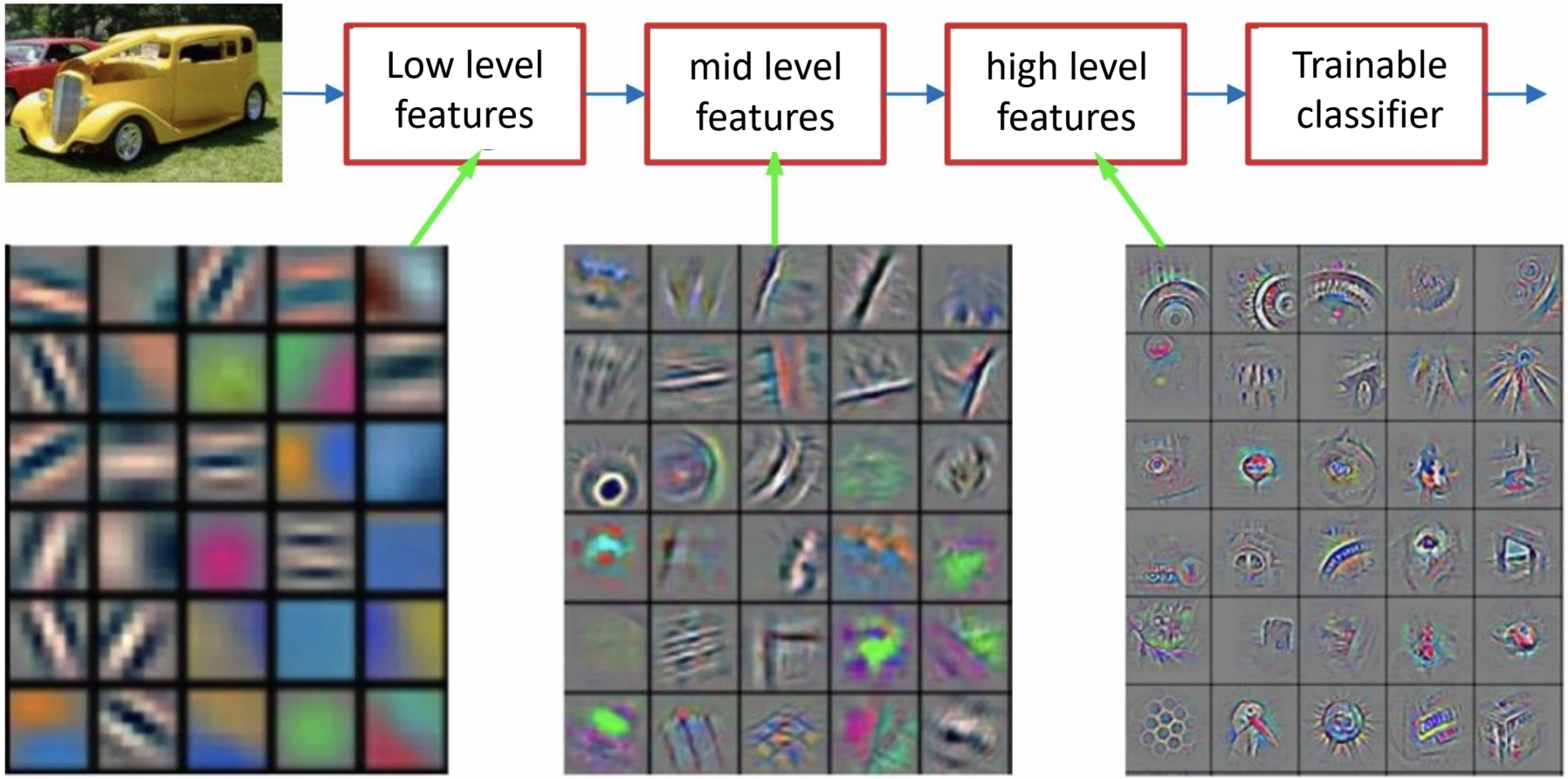


Convolutional Neural Networks

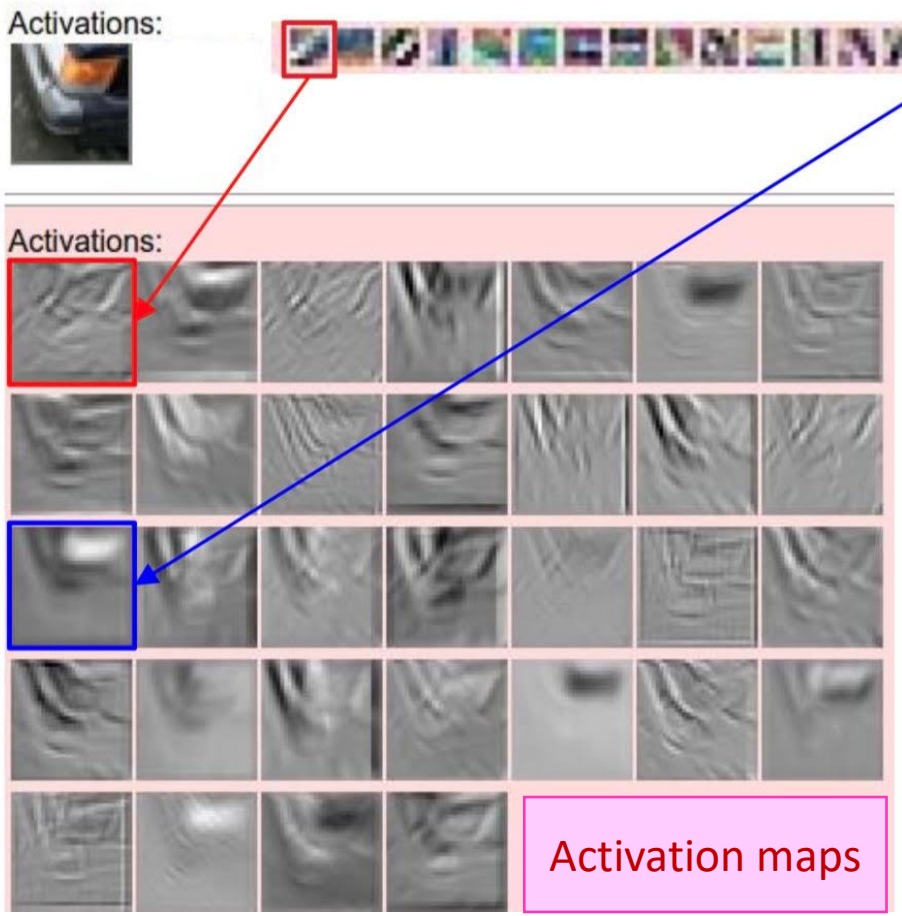
- Convolutional neural networks. A sequence of convolution layers and activity functions among them.



Convolutional Neural Networks



Convolutional Neural Networks



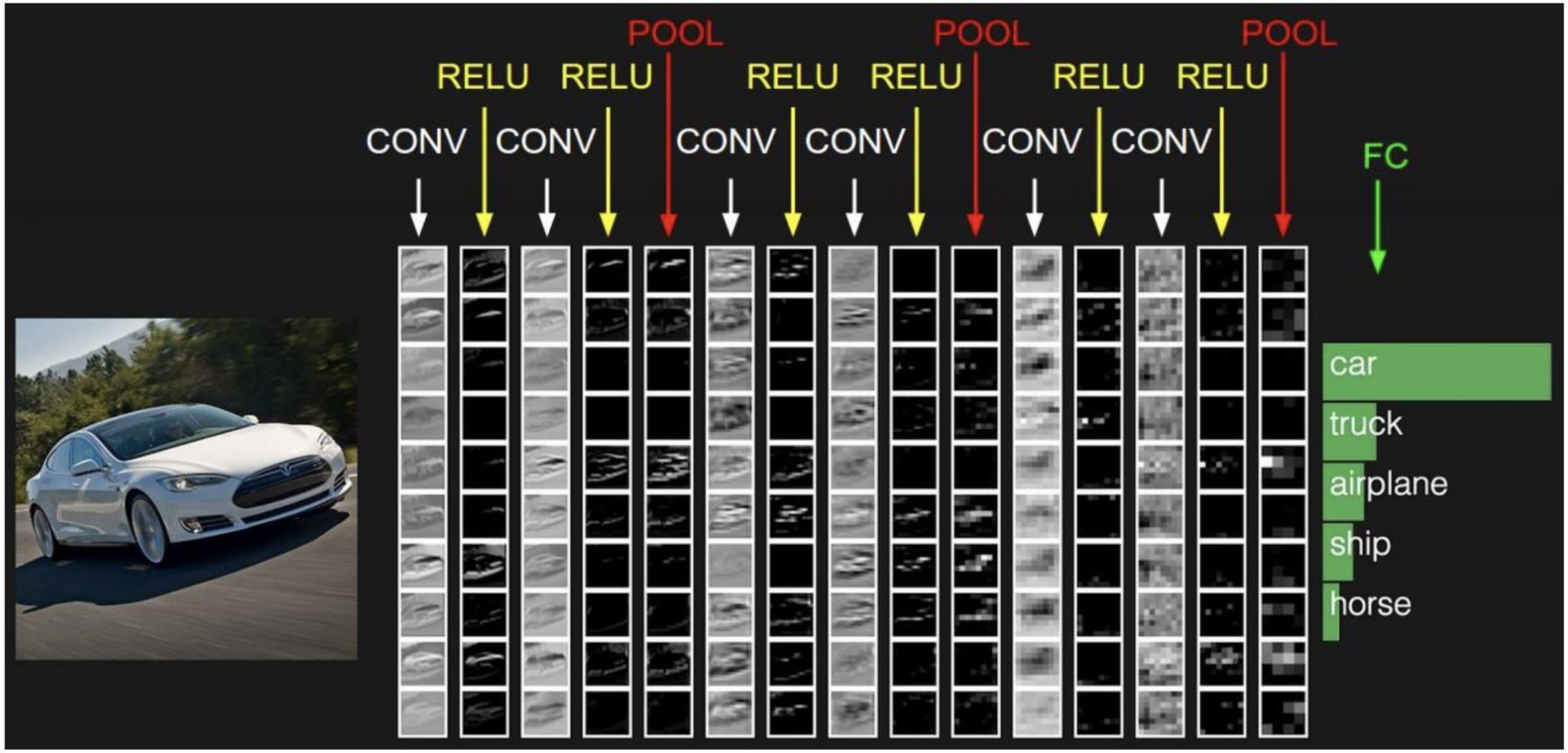
Sample filters with dimensions 5 x 5 (32 filters)

These layers are called convolution layers because they are related to the convolution of two signals:

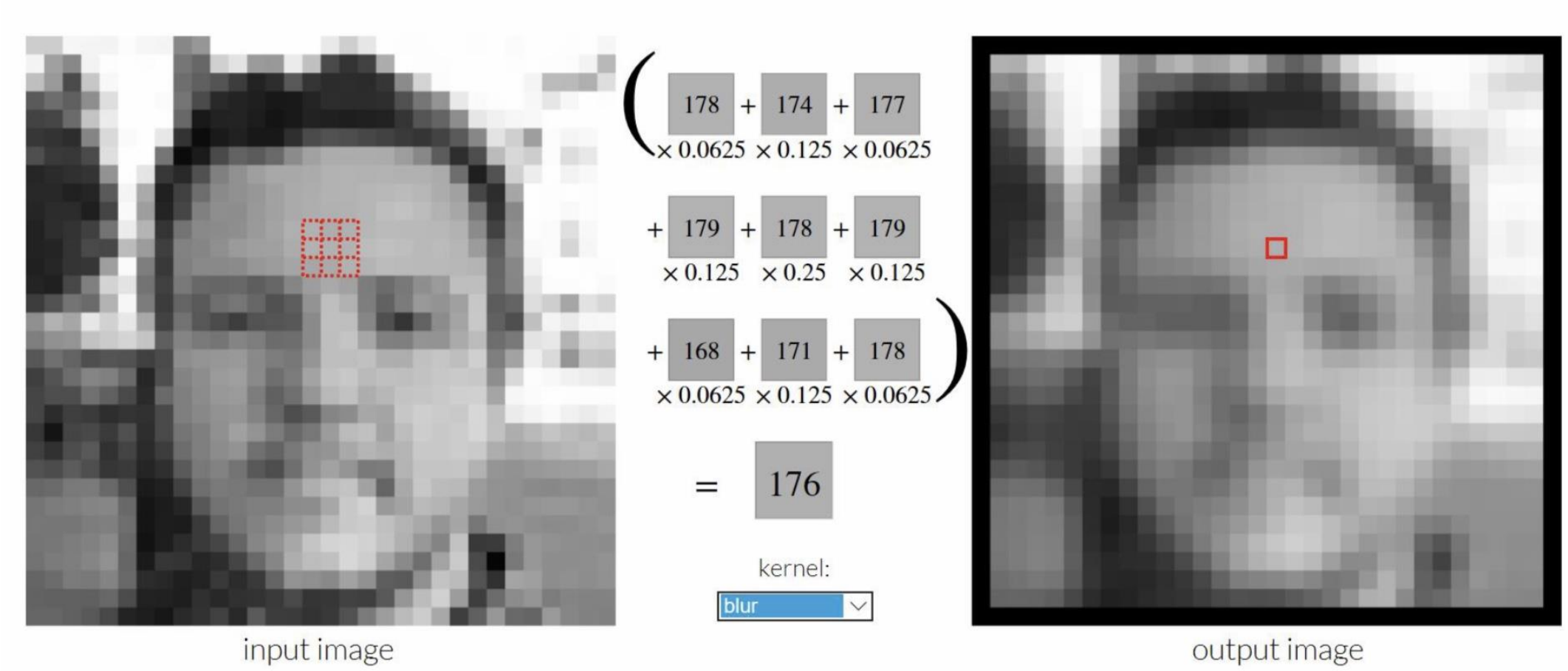
$$f[x,y] * g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1, y-n_2]$$

Multiply element by element and add a filter and signal

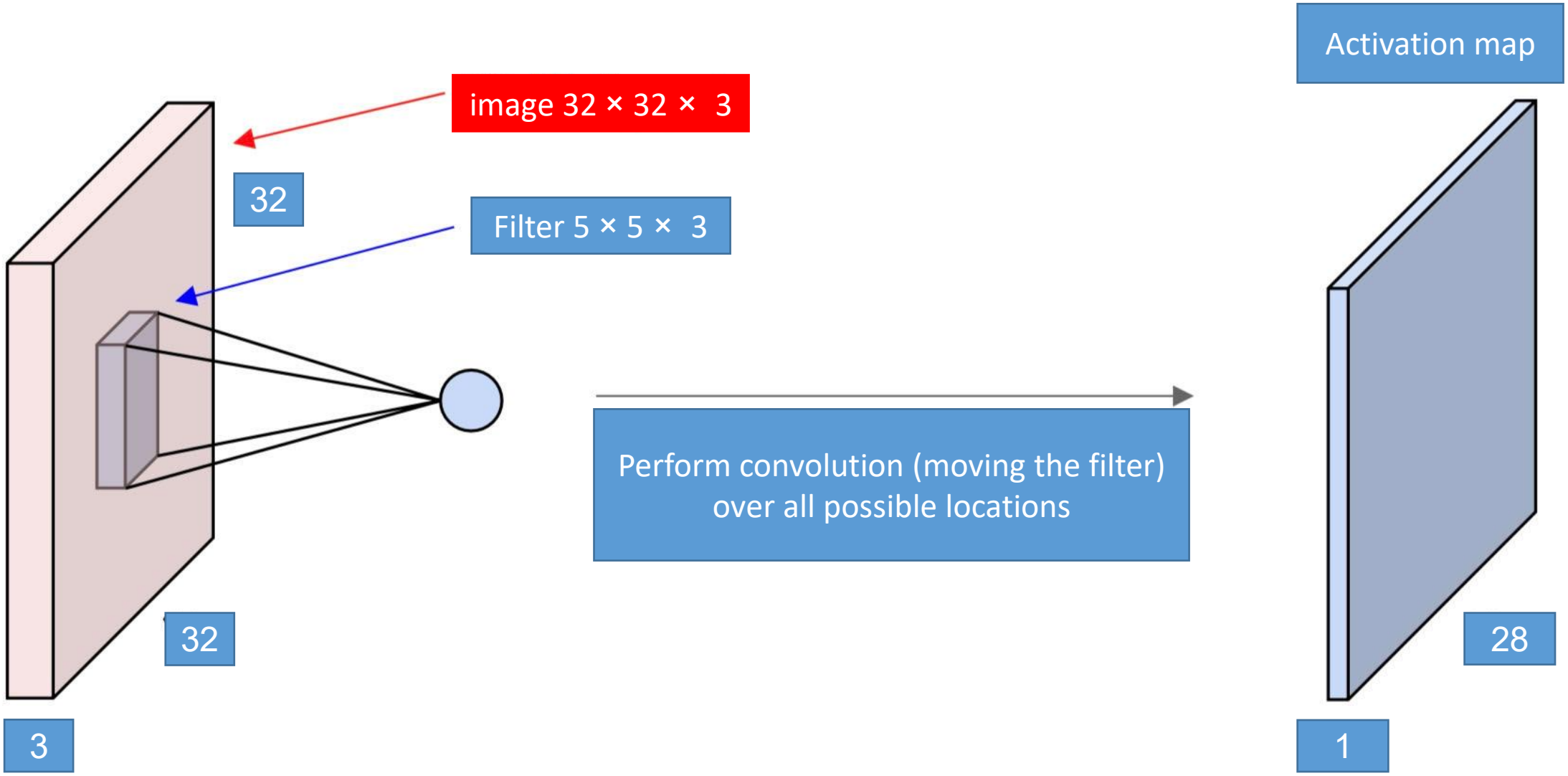
Convolutional Neural Networks



Convolution Layer: a closer look

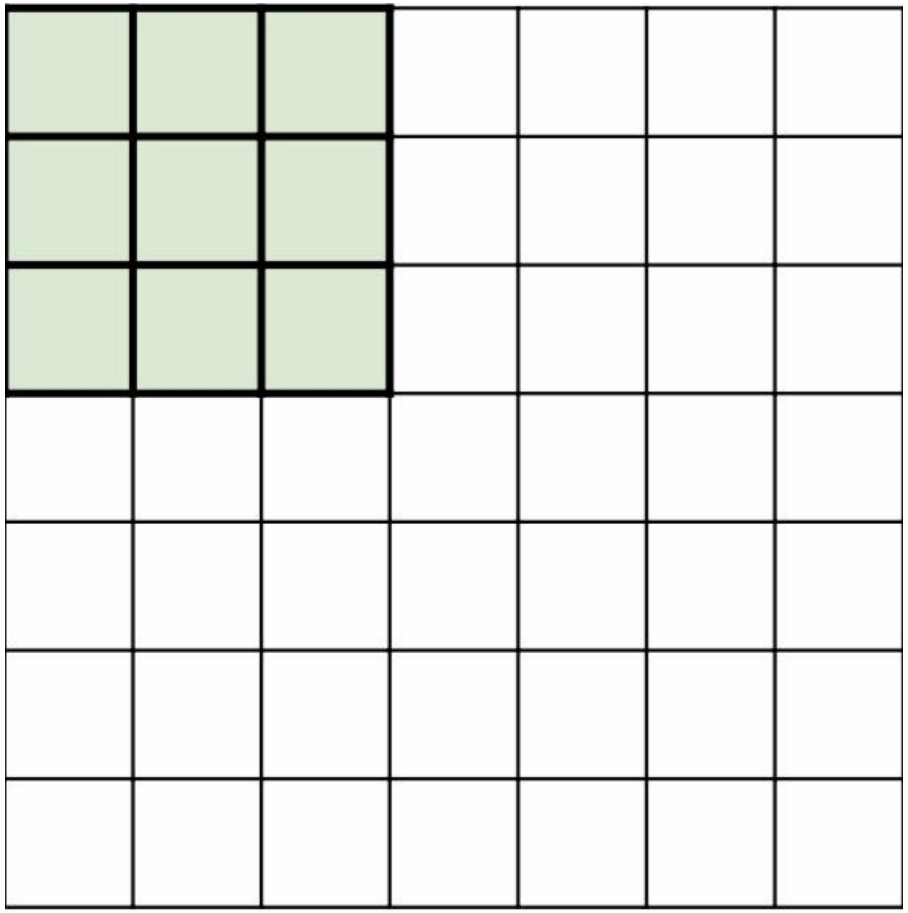


Convolution Layer: a closer look



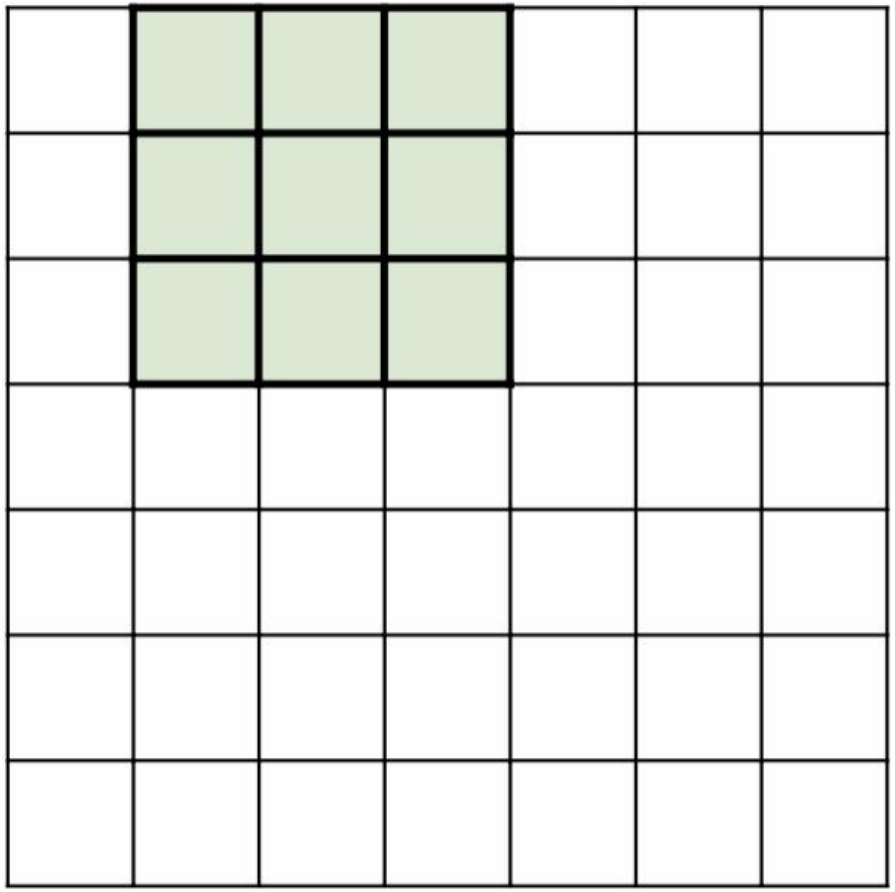
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3



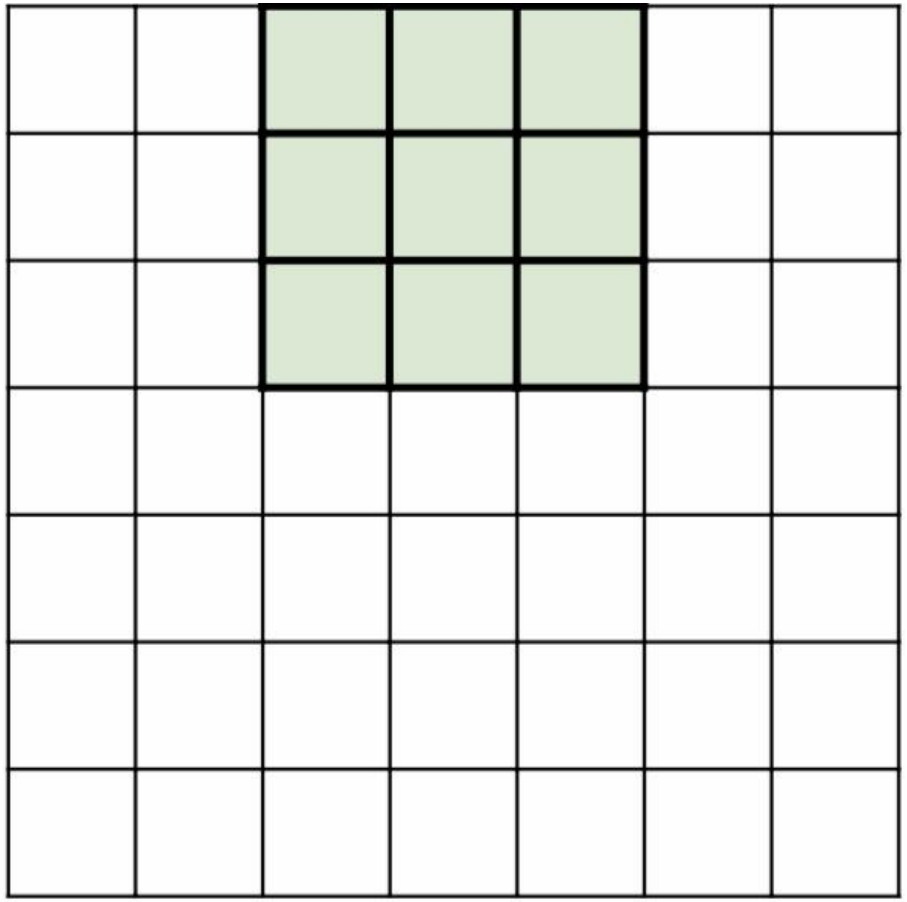
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3



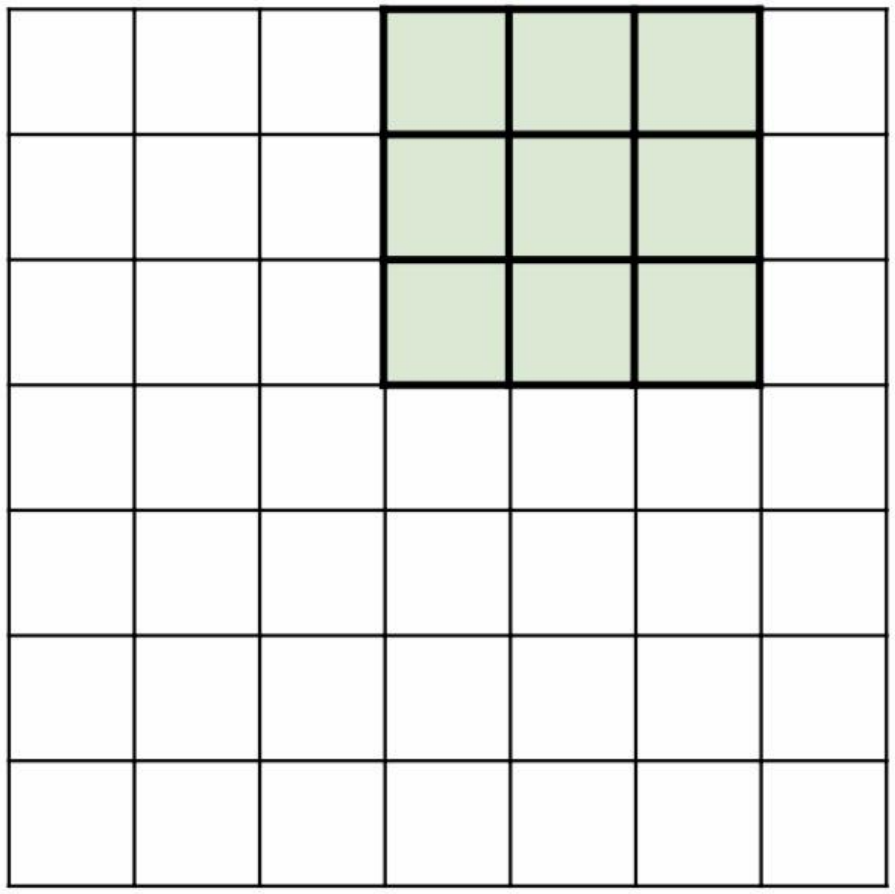
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3



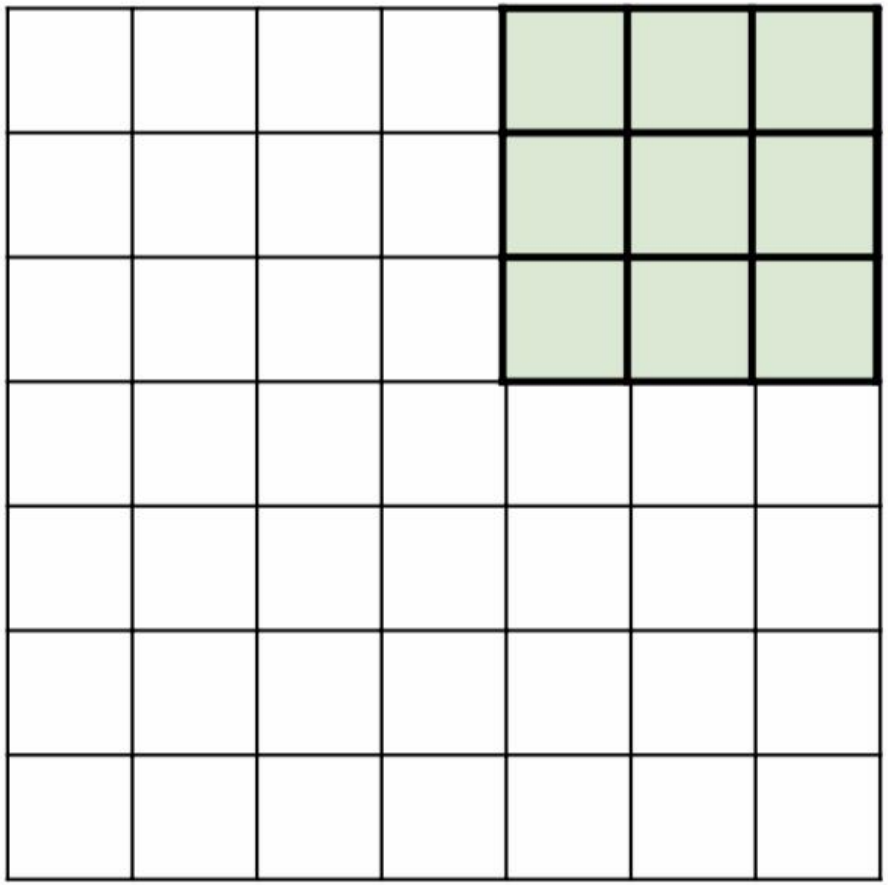
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3



Convolution Layer: a closer look

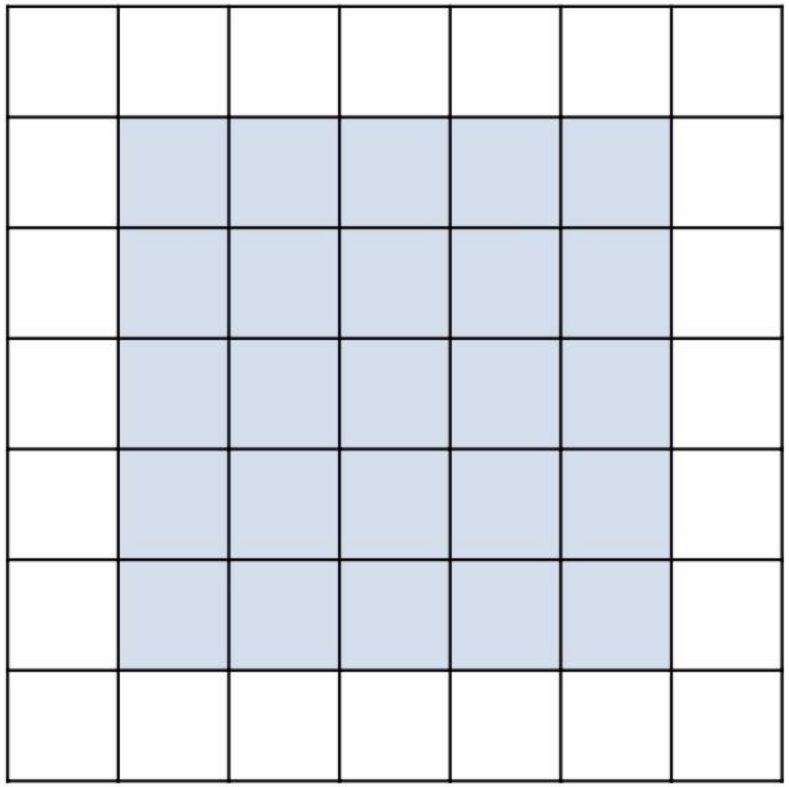
Input 7 × 7
Filter 3 × 3



Convolution Layer: a closer look

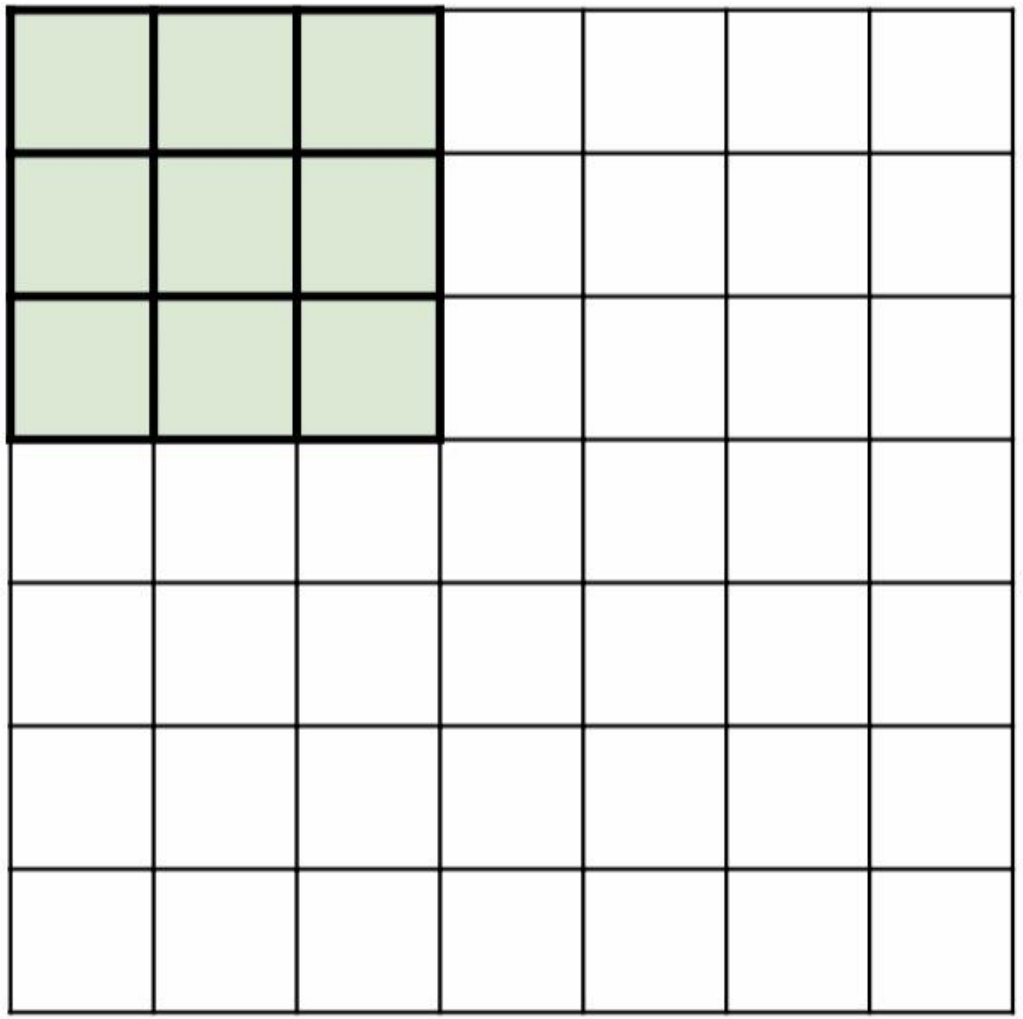
Input 7 × 7
Filter 3 × 3

Output 5 × 5



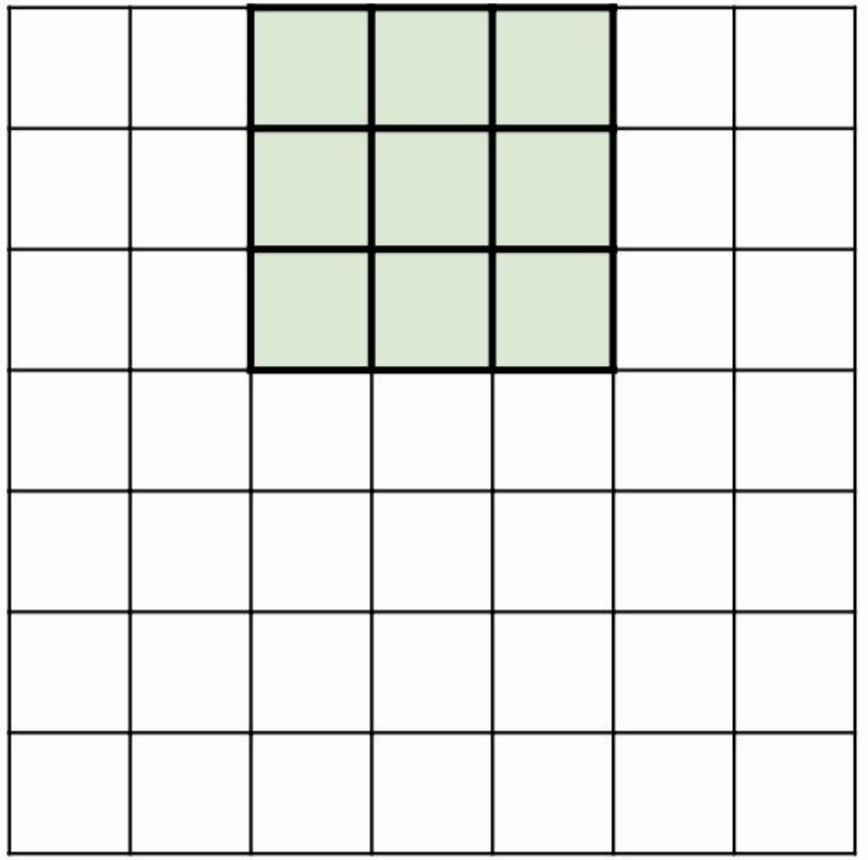
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3
Step size: 2



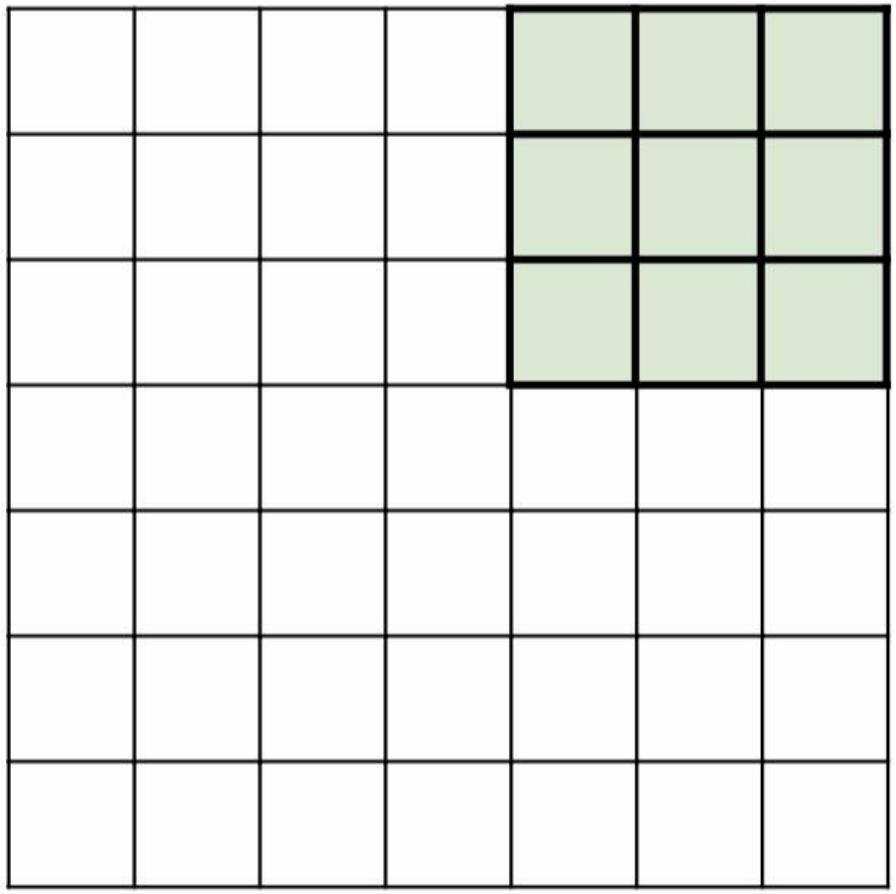
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3
Step size: 2



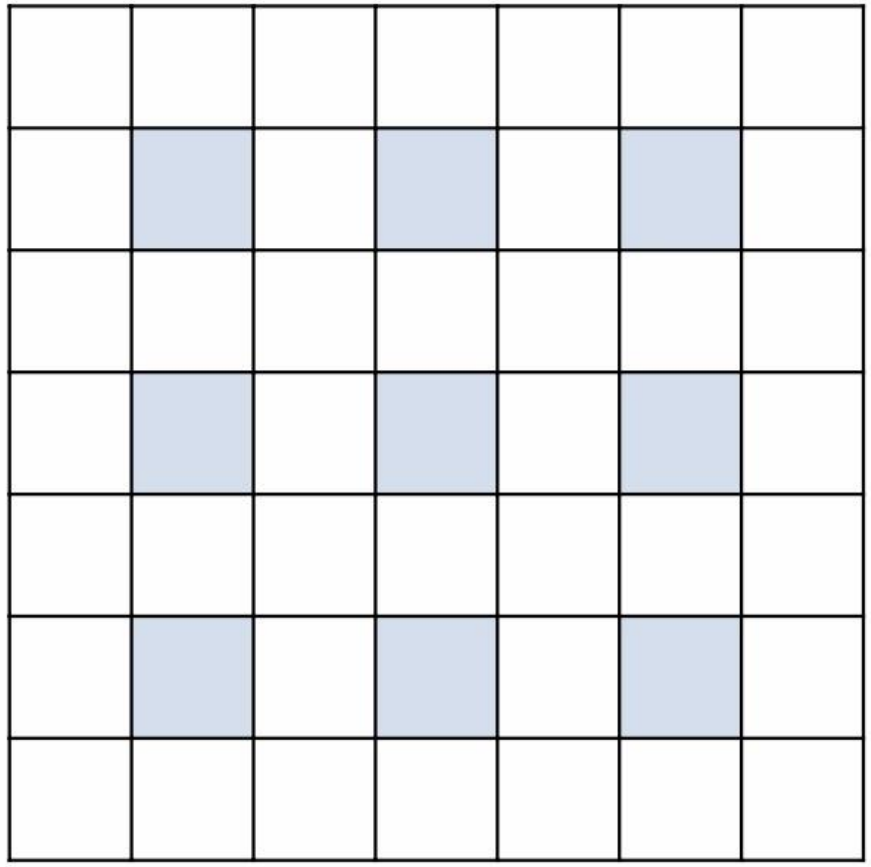
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3
Step size: 2



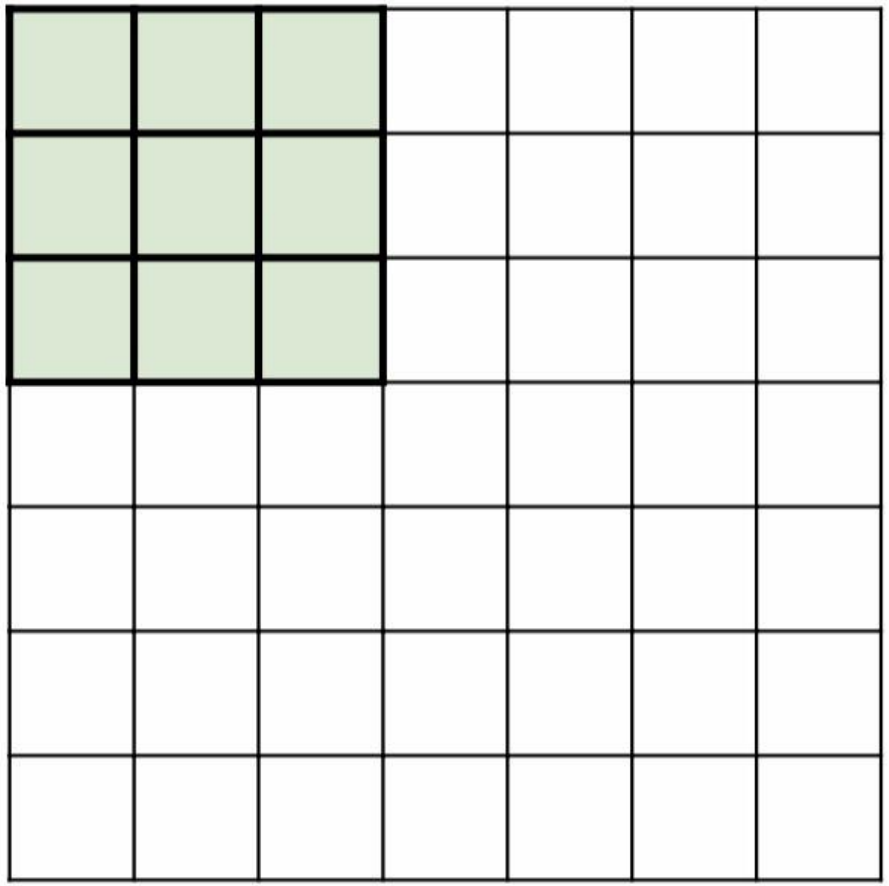
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3
Step size: 2
Output 3 × 3



Convolution Layer: a closer look

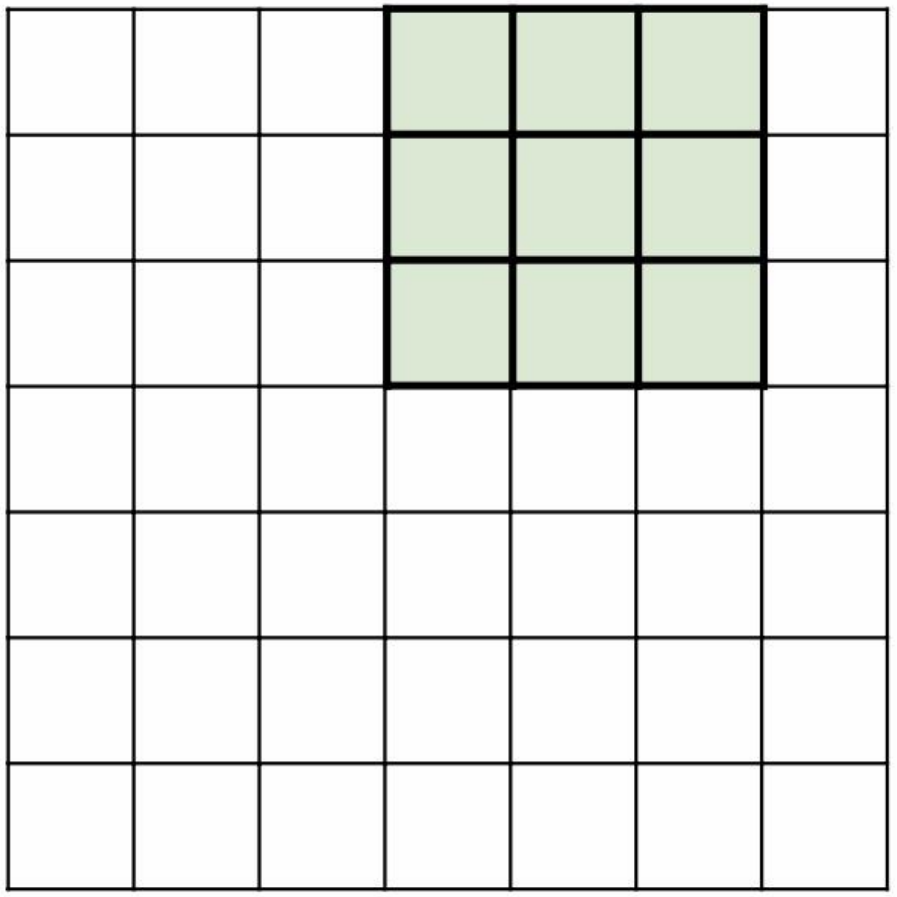
Input 7 × 7
Filter 3 × 3
Step size: 3



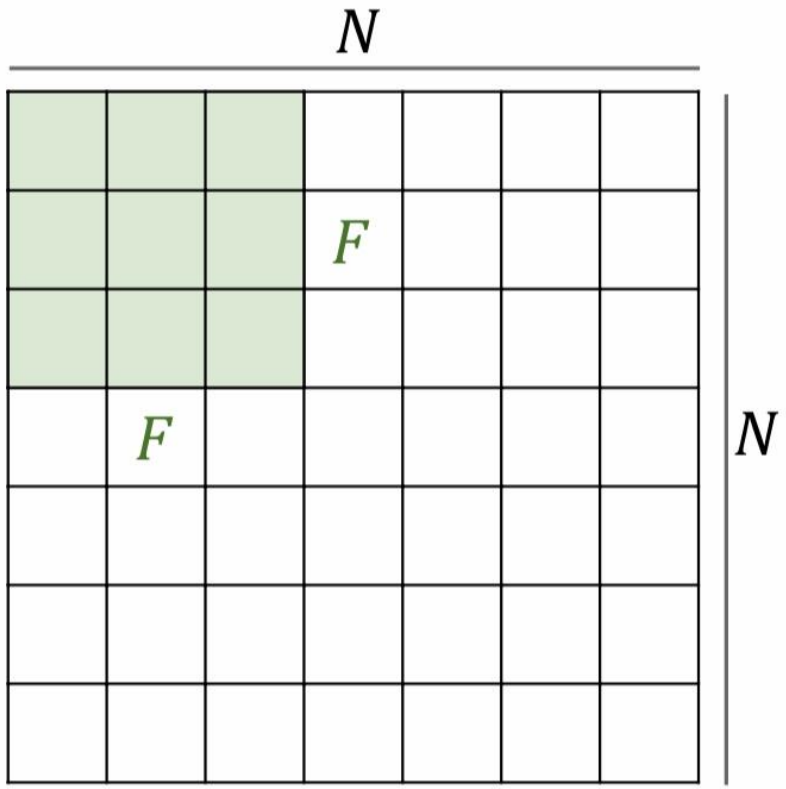
Convolution Layer: a closer look

Input 7 × 7
Filter 3 × 3
Step size: 2

It is not possible!



Convolution Layer: a closer look



Output size:

(N - F) / stride + 1

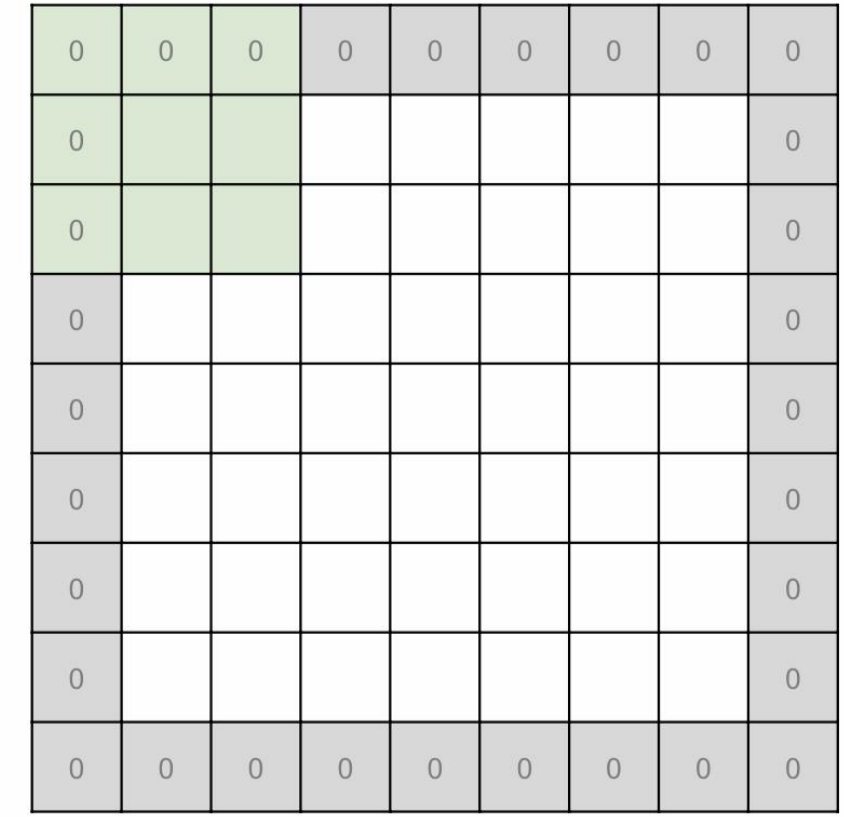
N = 7, F = 3

stride 1 => (7 - 3) / 1 + 1 = 5
stride 2 => (7 - 3) / 2 + 1 = 3
stride 3 => (7 - 3) / 3 + 1 = 2.33

Convolution Layer: padding

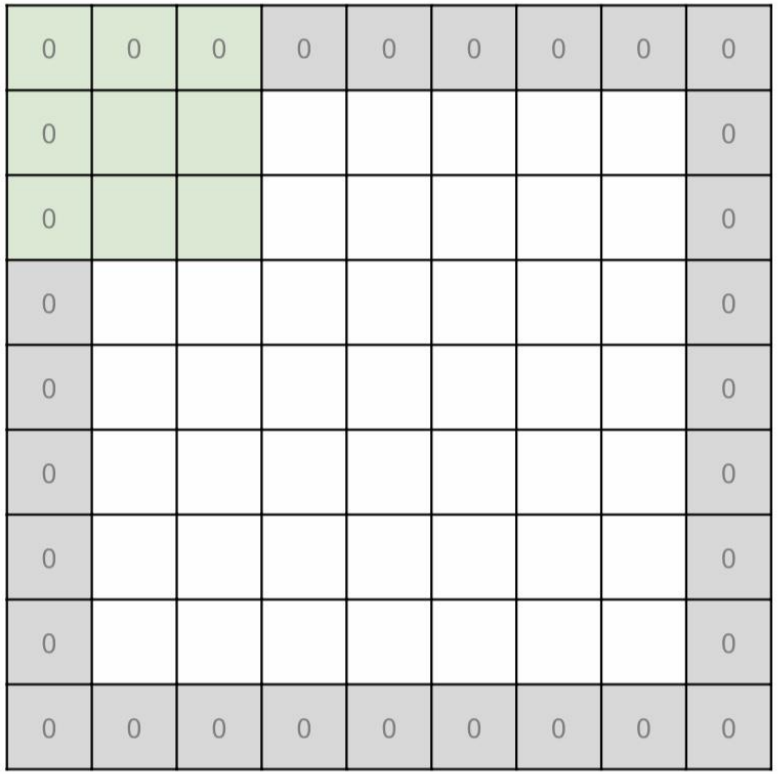
- Example: 7 x 7 image
- 3 × 3 fitler, with a step size of 1
- Expand borders by 1 pixel. output size?

Output 7 × 7



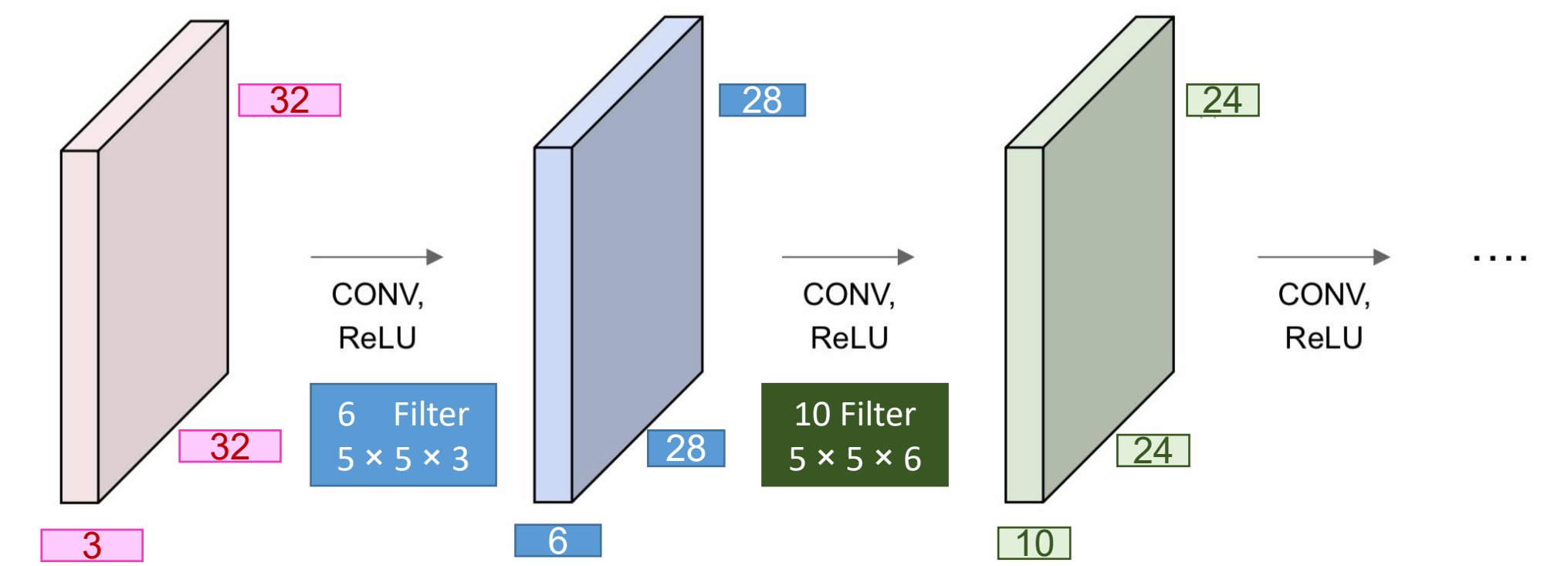
Convolution Layer: padding

- In general, convolution layers with a step size of 1, a filter size of F × F, and boundary zeroing of size (F-1)/2 are common. (input size unchanged)
- Example:
  - 3 × 3 filter ☞ zeroing to size 1
  - 5x5 filter ☞ zeroing to size 2
  - 7x7 filter ☞ Zeroing to size 3



Reminder: Convolutional Neural Networks

- Convolutional neural networks. A sequence of convolution layers and activity functions among them.
- Downsizing too quickly is not a good idea! (32 ☞ 28 ☞ 24 ☞ …)

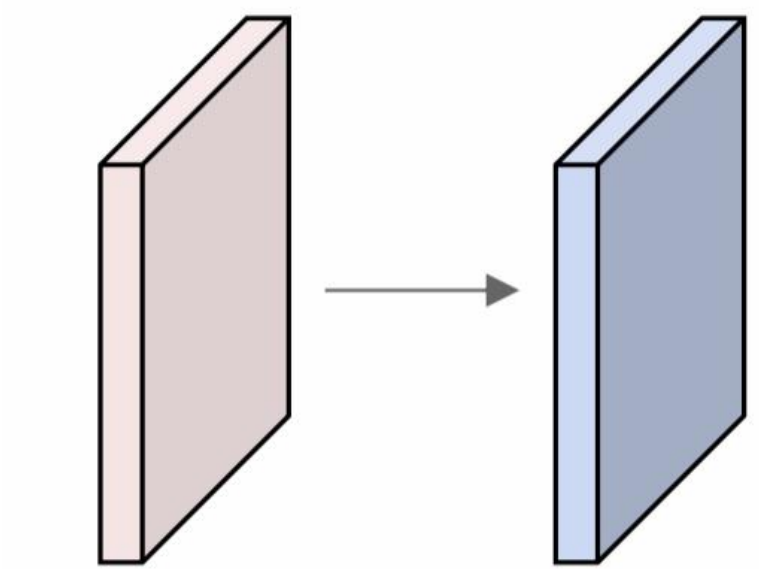


Convolution Layer: example

- Input size: 32 x 32 x 3
- 10 5 x 5 filters, with 1 step and 2 pixel zeroing
- output size?

$(32 - 5 + 2 \times 2) / 1 + 1 = 32$ ← The size of each Activation map

$32 \times 32 \times 10$ ← The number of Activation maps



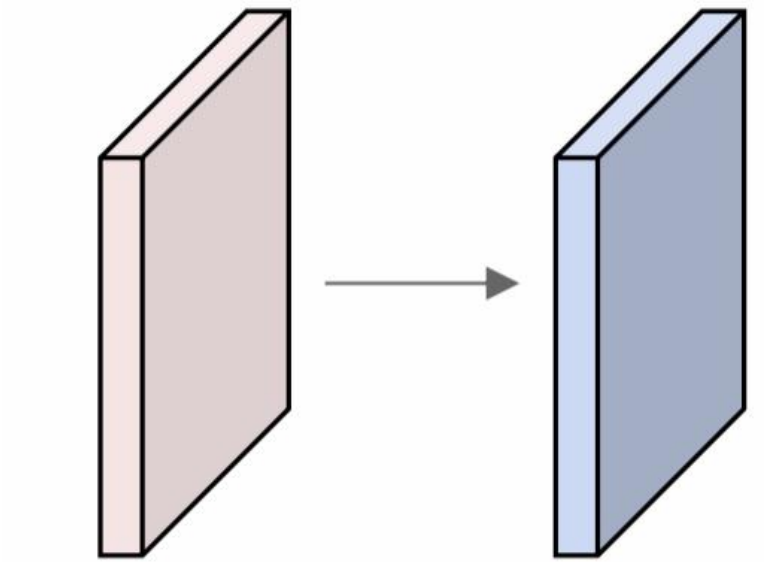
Convolution Layer: example

- Input size: 32 x 32 x 3
- 10 5 x 5 filters, with 1 step and 2 pixel zeroing
- Total number of parameters in this layer?

For bias

$5 \times 5 \times 3 + 1 = 76$ ← The number of parameters for each filter

$76 \times 10 = 760$ ← Total number of parameters



Convolution Layer: Conclusion

Common values:

K = power of 2 (such as 32, 64, 128, and 512)
P = 1, S = 1, F = 3
P = 2, S = 1, F = 5
P = ?, S = 2, F = 3 (any suitable value)
P = 0, S = 1, F = 1

✓Get an input with $W_1 \times H_1 \times D_1$ Dimensions

✓ Need four hyperparameters:

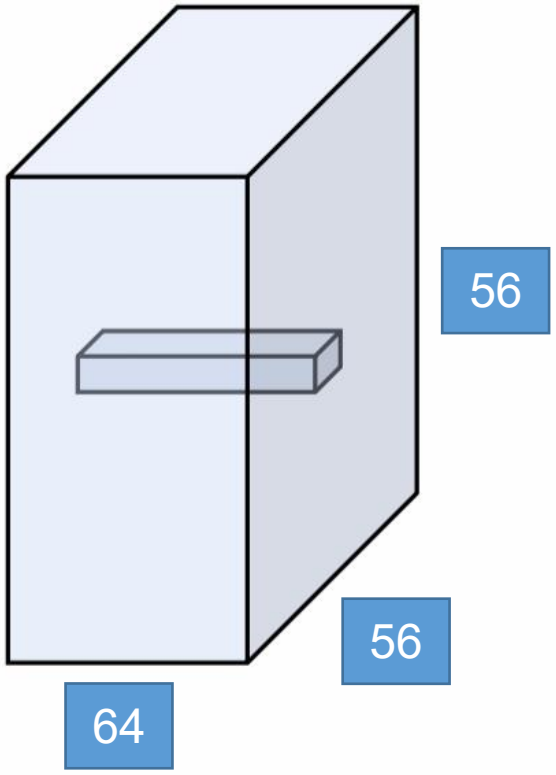
- The number of filters is K
- The dimensions of each filter are F × F
- Step size S
- The zeroing rate of P

✓ Produce an output with $W_2 \times H_2 \times D_2$ dimensions, so that:

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$
- $D_2 = K$

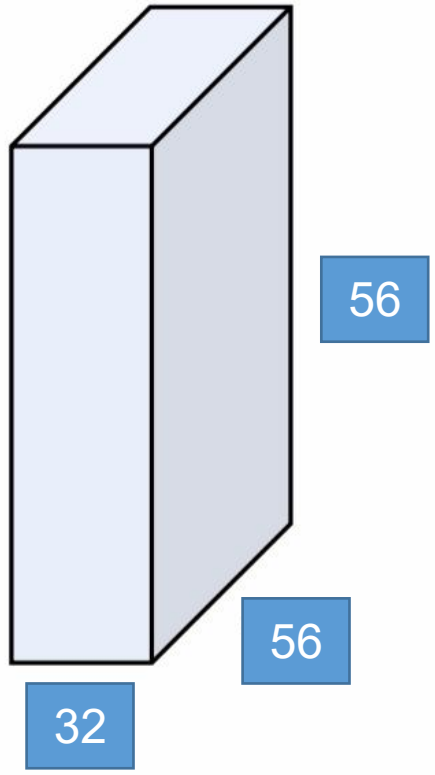
✓ It has F × F × D_1 weights per filter and total number of (F × F × D_1) × K weights and K bias.

Convolution Layer: with 1 × 1 filters

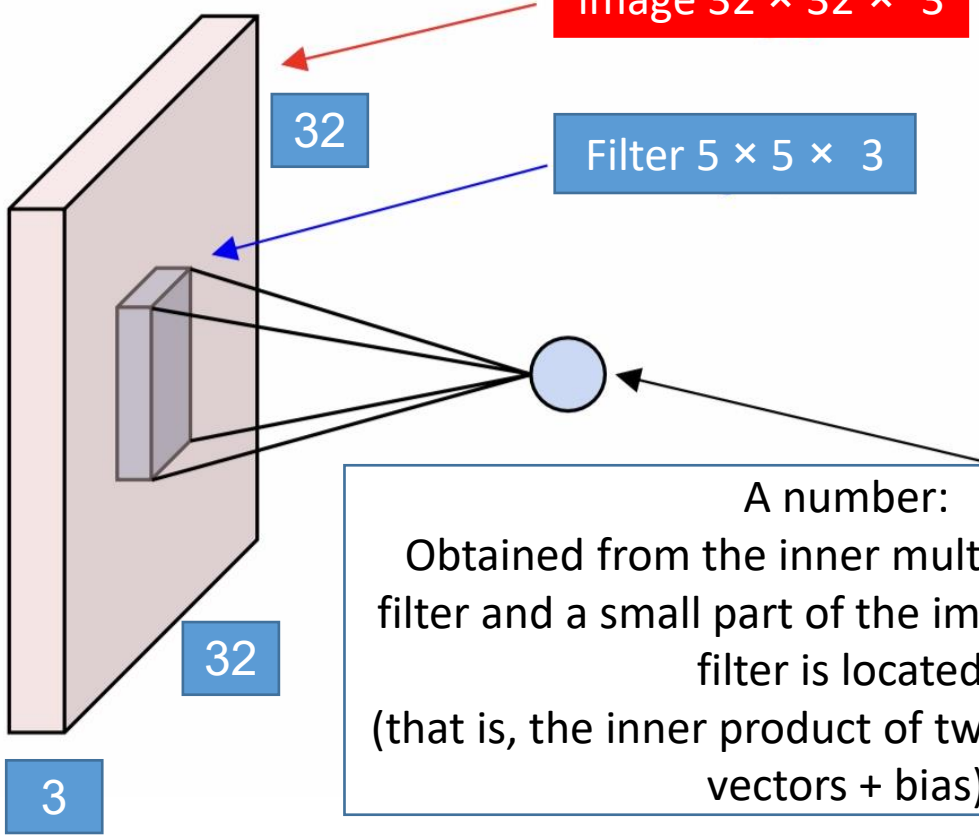


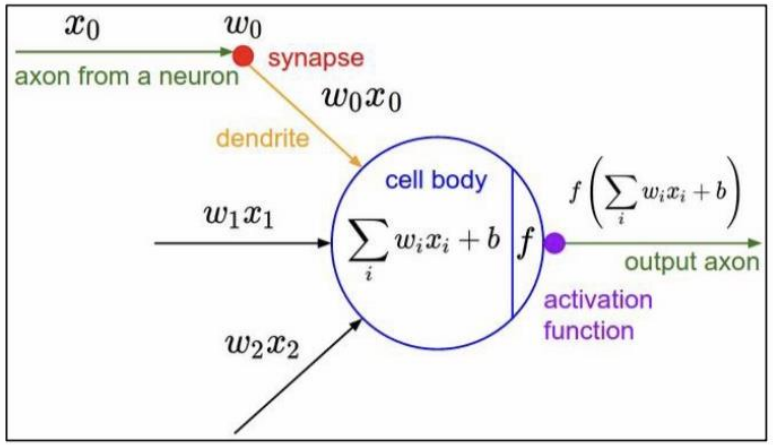
Convolution Layer with 32 numbers of 1 × 1 filter

Each filter is equivalent to the inner product of two 64-dimensional vectors



Neuron's view of the convolution layer

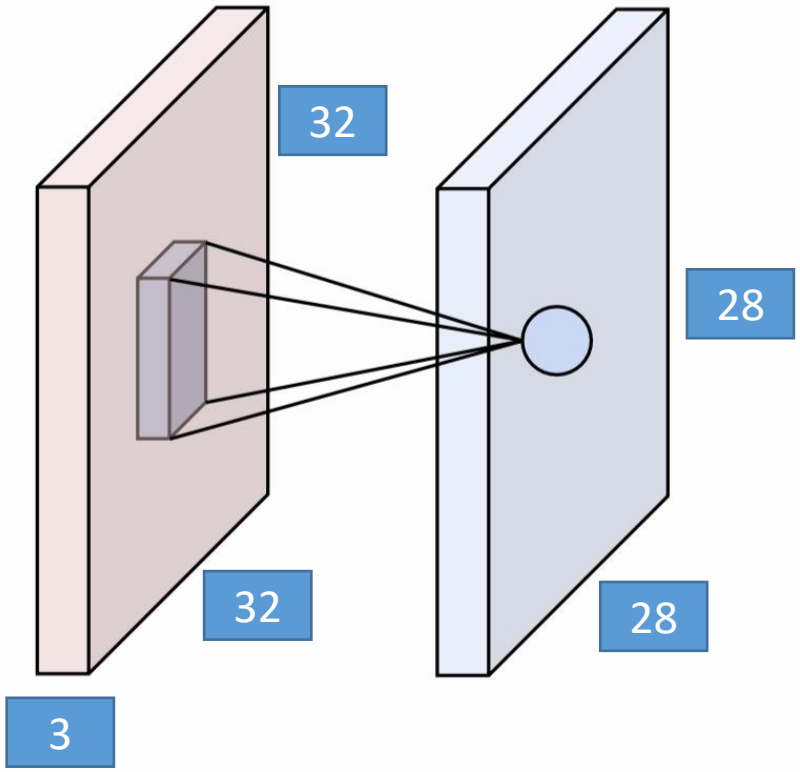


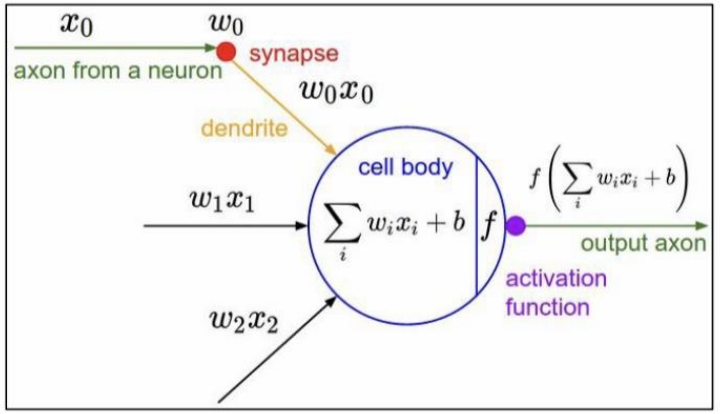


A neuron with local connections

A number:
Obtained from the inner multiplication of the filter and a small part of the image on which the filter is located.
(that is, the inner product of two 75-dimensional vectors + bias)

Neuron's view of the convolution layer

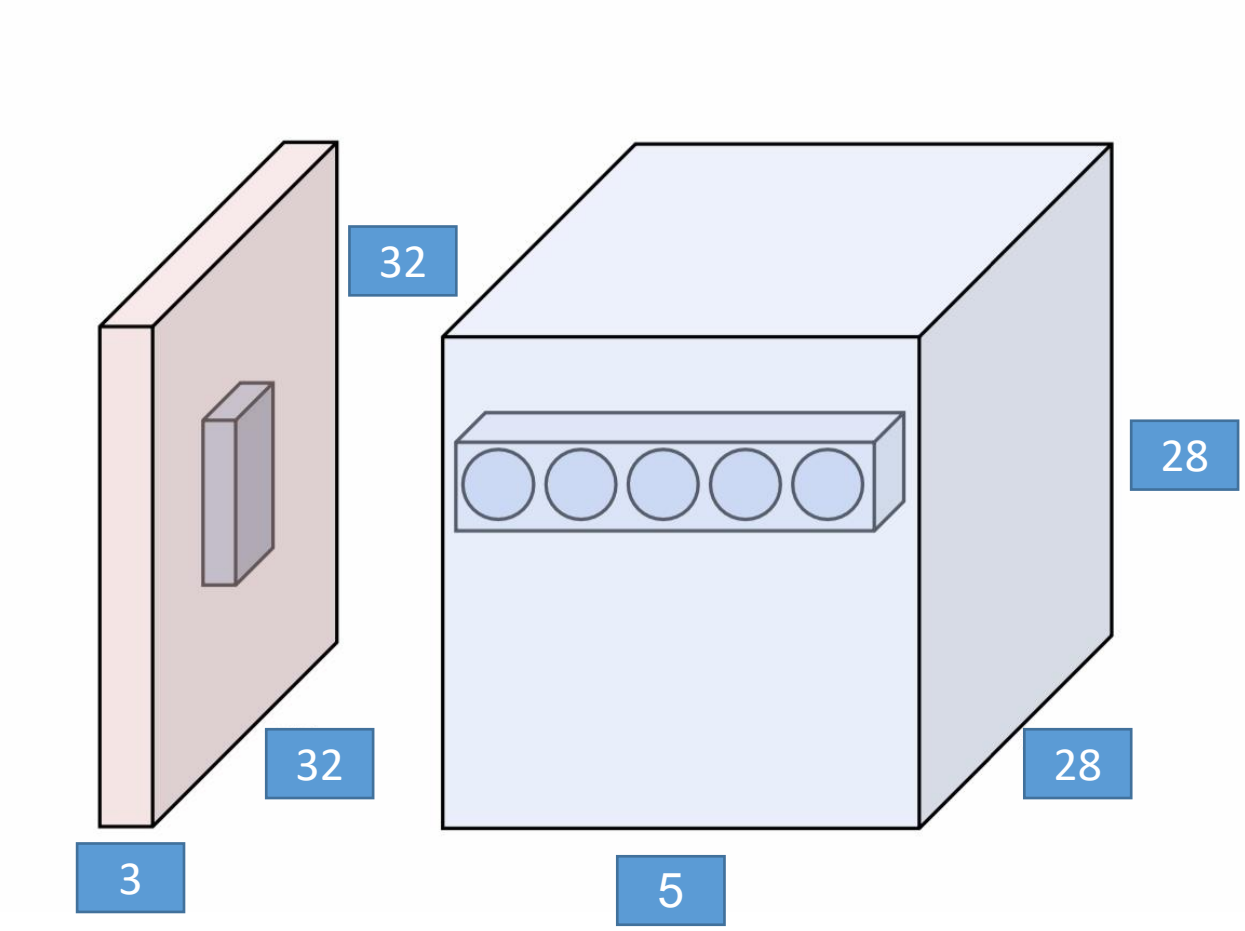


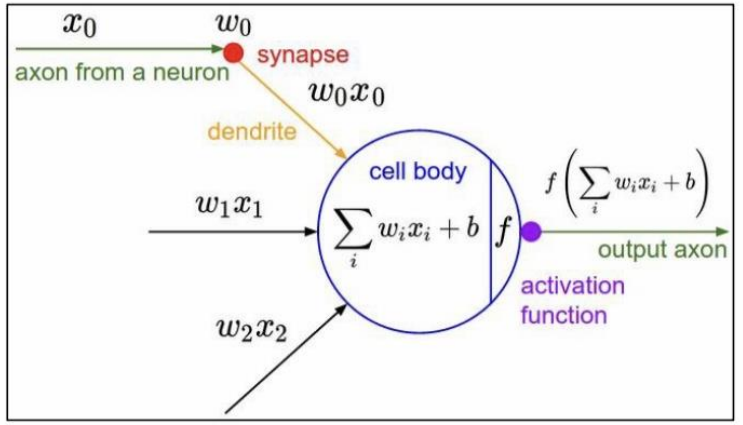


Each activity map is a 26 x 26 layer of neuron output:
1. Each neuron is connected to a small area of input.
2. All neurons have common parameters.

"5 x 5 filter": a 5 x 5 field of view for each neuron!

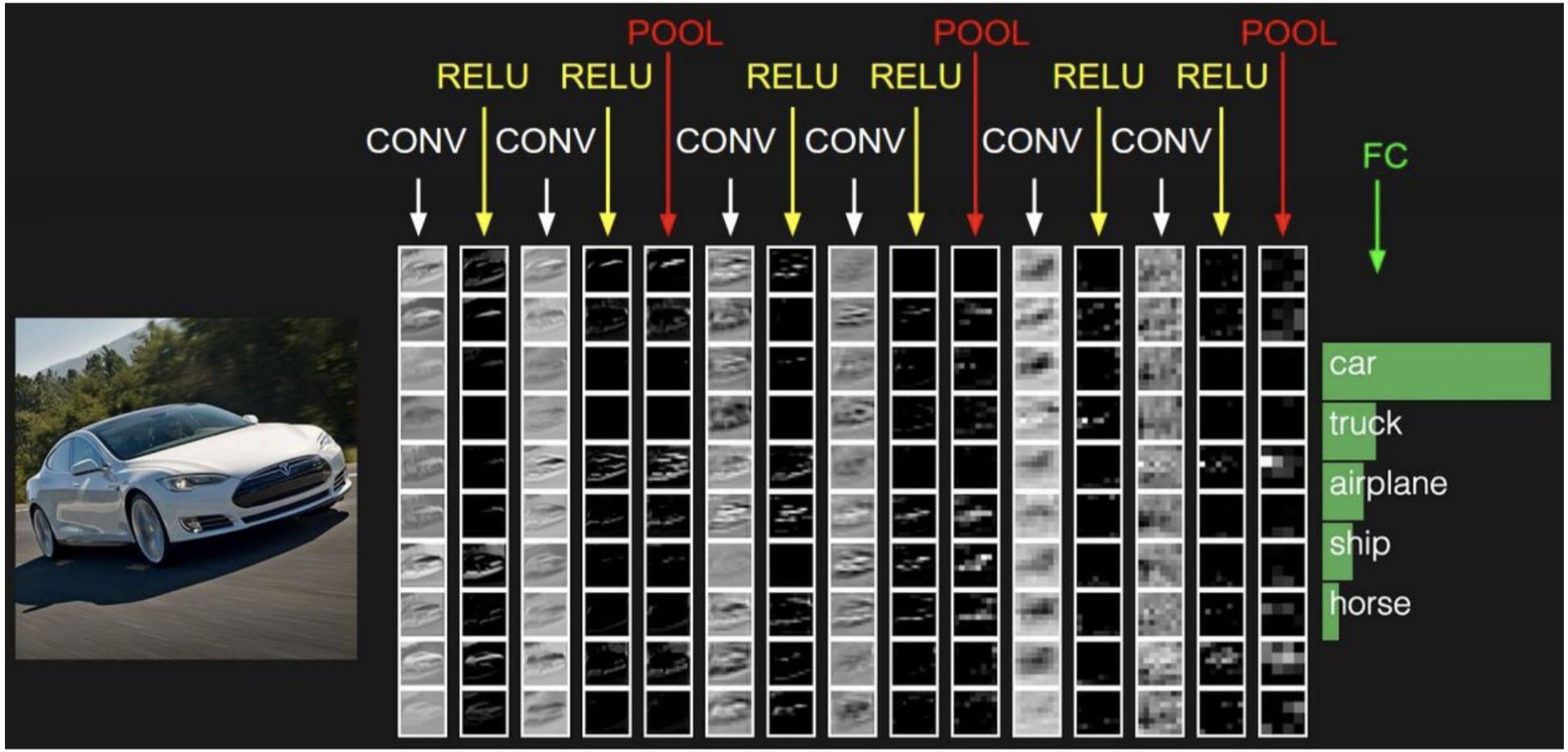
Neuron's view of the convolution layer





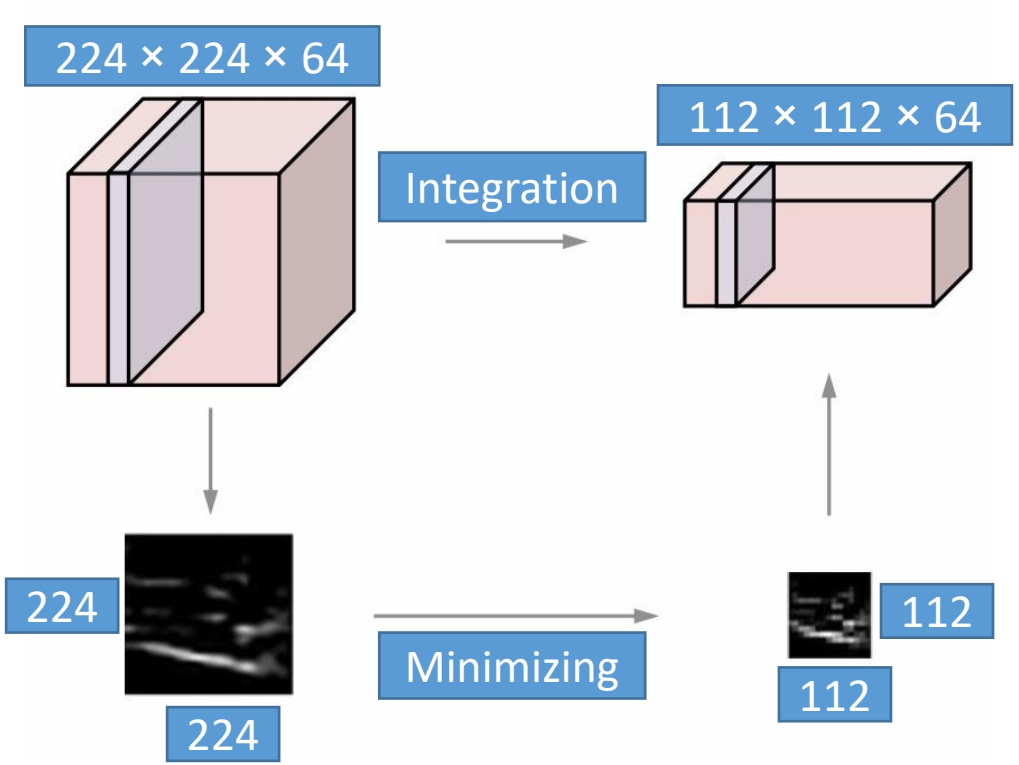
For example, with 5 filters, the convolution layer consists of a number of neurons arranged in a 3-dimensional grid.
(28 x 28 x 5)

Convolutional Neural Networks: Merging Layers



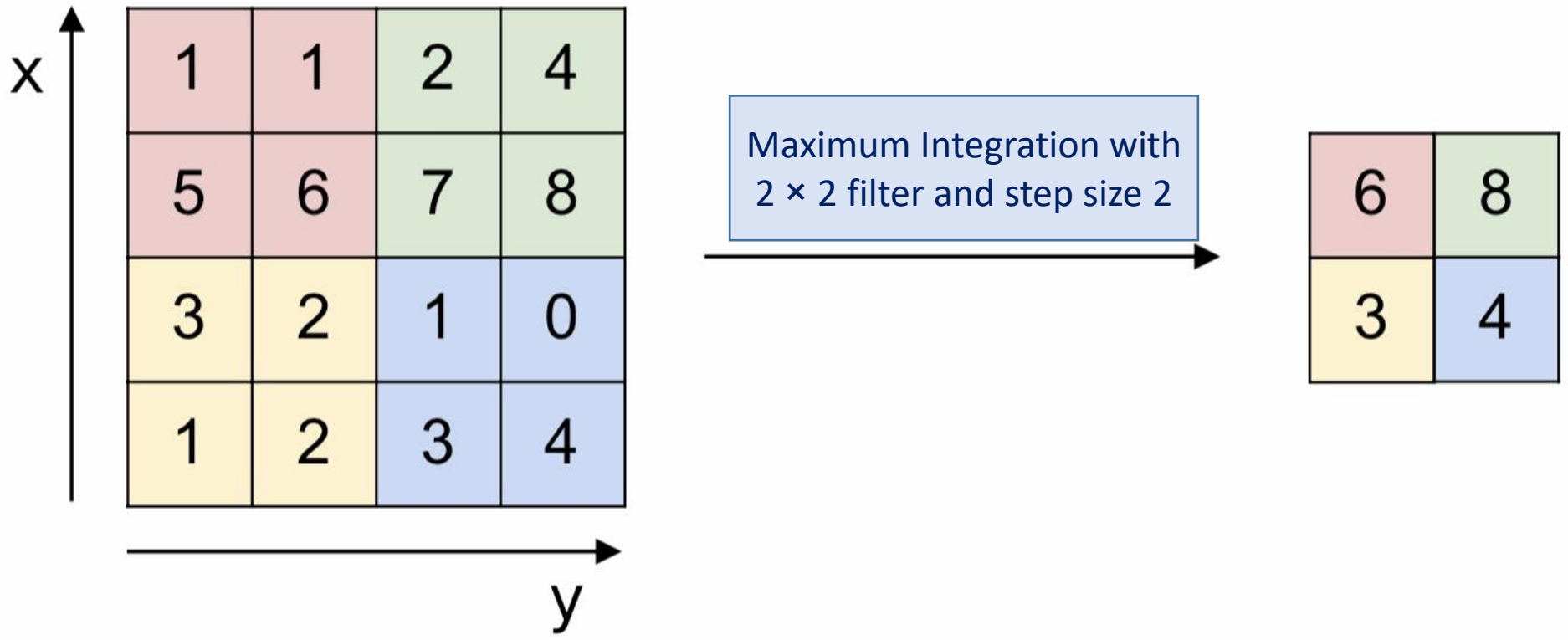
Convolutional Neural Networks: Merging Layers

- Integration layer. Minimize the representation!
  - It works on each activity map separately.



Convolutional Neural Networks: Maximum Integration

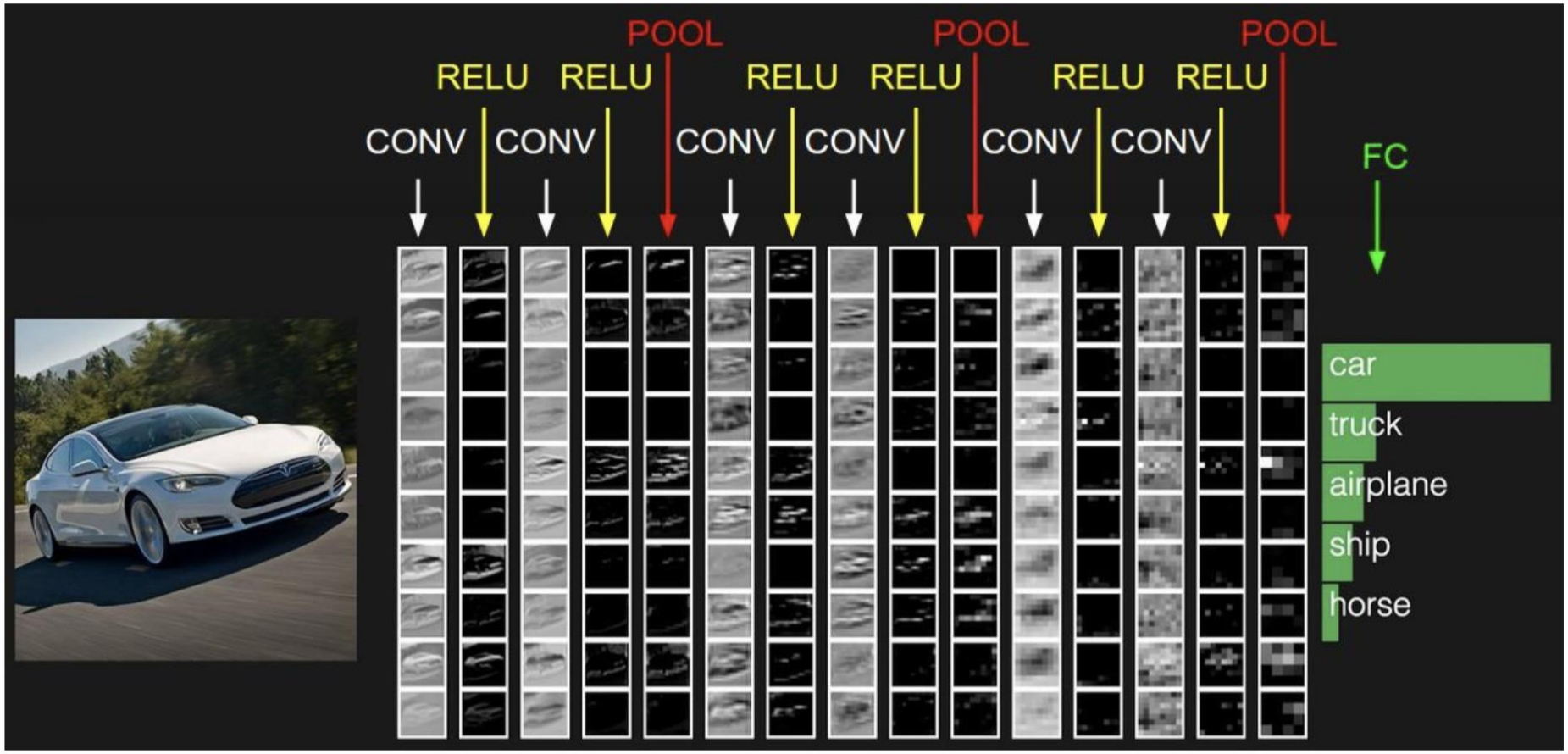
- Integration layer. Minimize the representation!
  - It works on each activity map separately.



Integration layer: Summarization

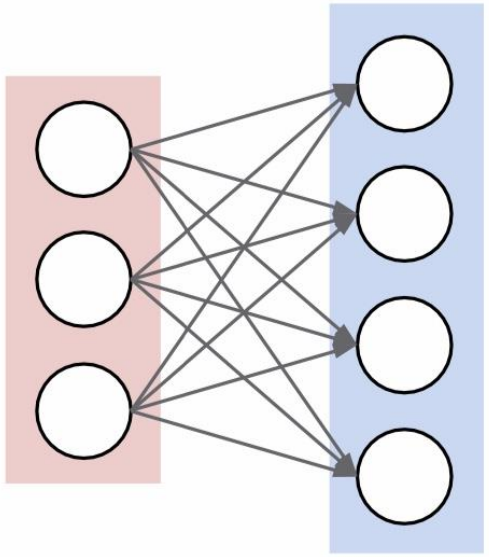
- Receive an input with dimensions w1 x h1 x d1
- Need three hyper parameters:
  - Dimensions of each F × F filter (such as 2 or 3)
  - Step size S
- Produce an output with dimensions W2 × H2 × D2, so that:
  - W2 = (W1 - F)/S + 1
  - H2 = (H1 - F)/S + 1
  - D2 = D1
- The number of parameters is equal to zero!
- Note: It is not common to use zeroing for merge layers.

Convolutional Neural Networks: Fully connected layers



Convolutional Neural Networks: Fully connected layers

- Fully connected layers: Contains neurons that are connected to all inputs!
  - Similar to the layers used in a normal neural network



Convolutional Neural Networks: Conclusion

- Convolution neural networks: a sequence of FC and POOL layers, CONV
- Trend towards smaller filters and deeper structures.
- Tendency towards complete removal of integration layers and fully connected layers.
- A common pattern for the structure of convolutional networks:

  [(CONV-RELU)*N-POOL?]*M-(FC-RELU)*K,SOFTMAX

  - So that N is at most 5, M is a large value and K is between 0 and 2.
  - But recent developments such as ResNet and GoogleNet have challenged this model!