Introduction to Big Data Science

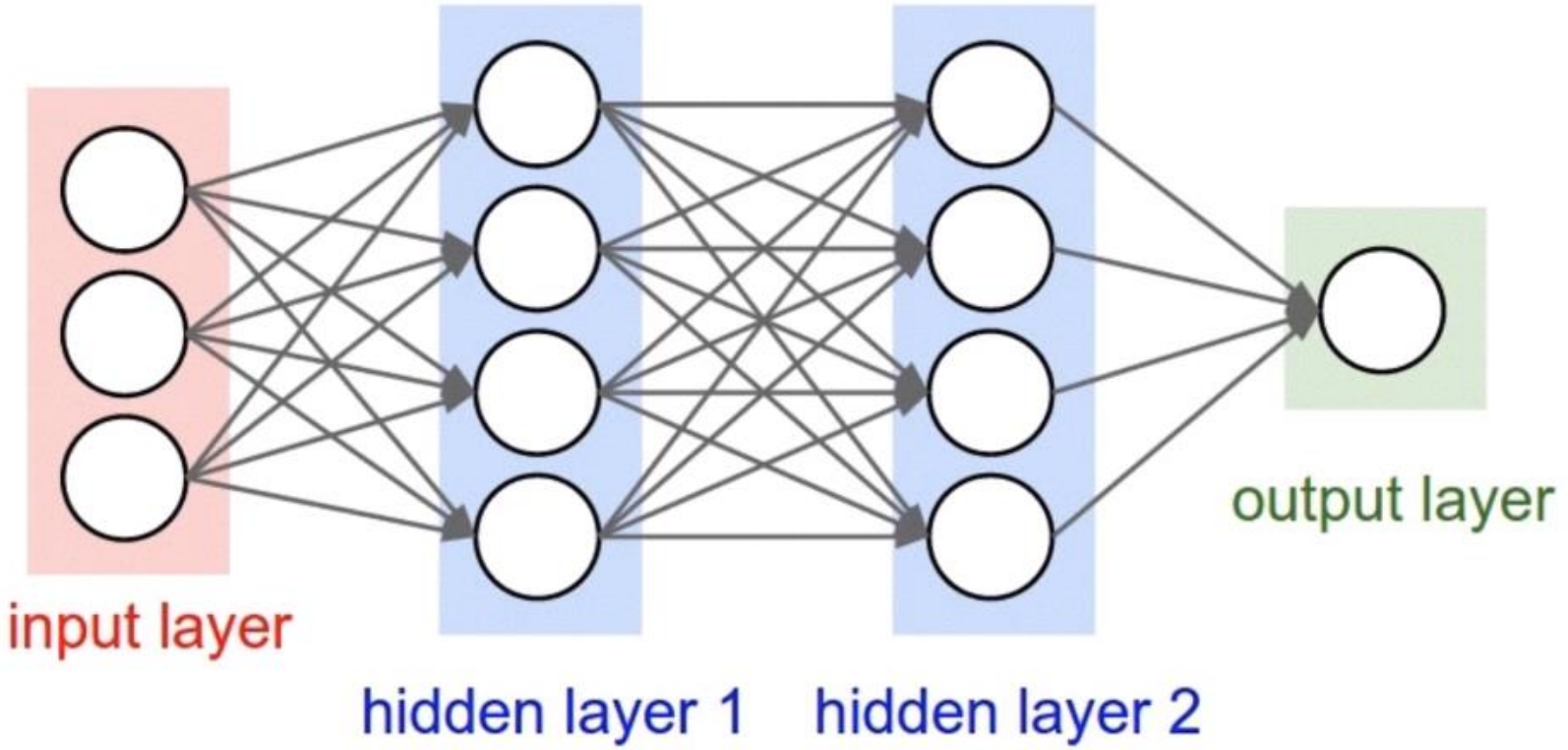
11-2 Period

Introduction to Deep Learning (Deep Belief Network, Drop-Out, Convolution Neural Network)

Contents

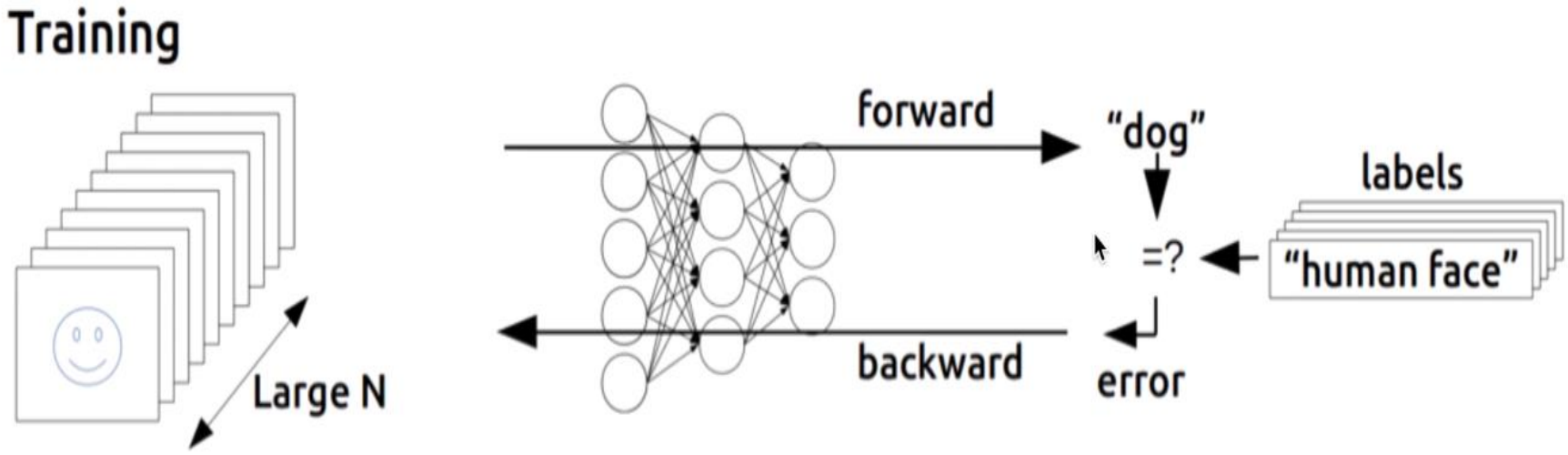
- **Deep Belief Network**
- **Drop Out**
- **Convolutional Neural Network**

Derivation



Back Propagation

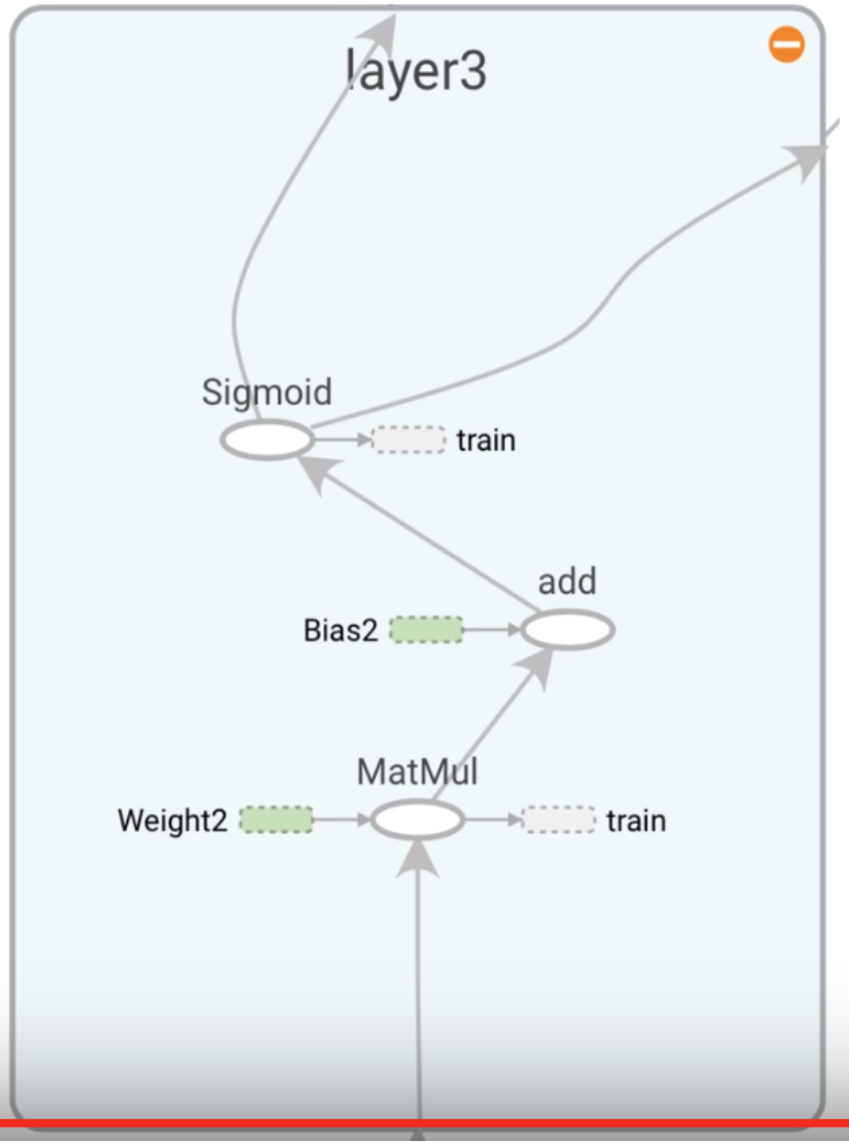
(1974, 1982 by Paul Werbos, 1986 by Hinton)



Neural Network Principle

Go to Neural Network Slide

BP in TensorFlow

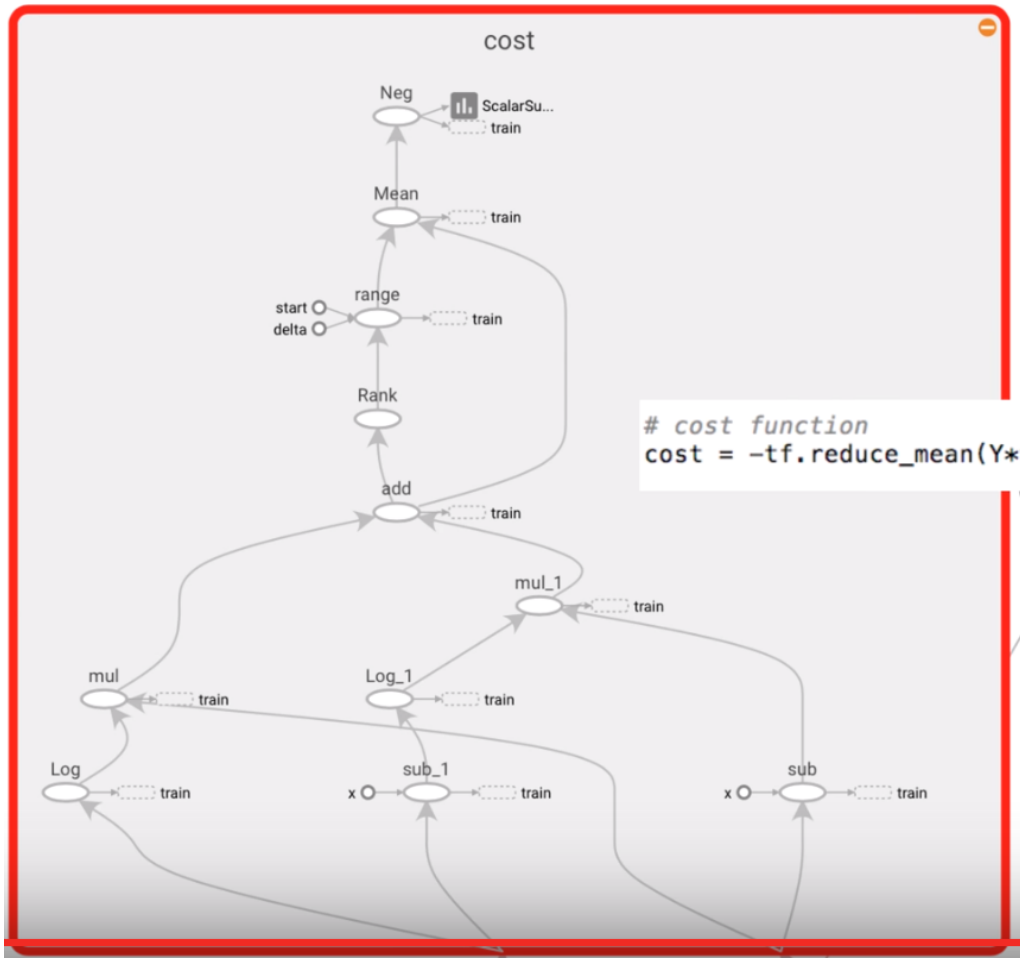


Back propagation in TensorFlow

TensorBoard

```
hypothesis = tf.sigmoid(tf.matmul(L2, W2) + b2)
```

BP in TensorFlow (TB)

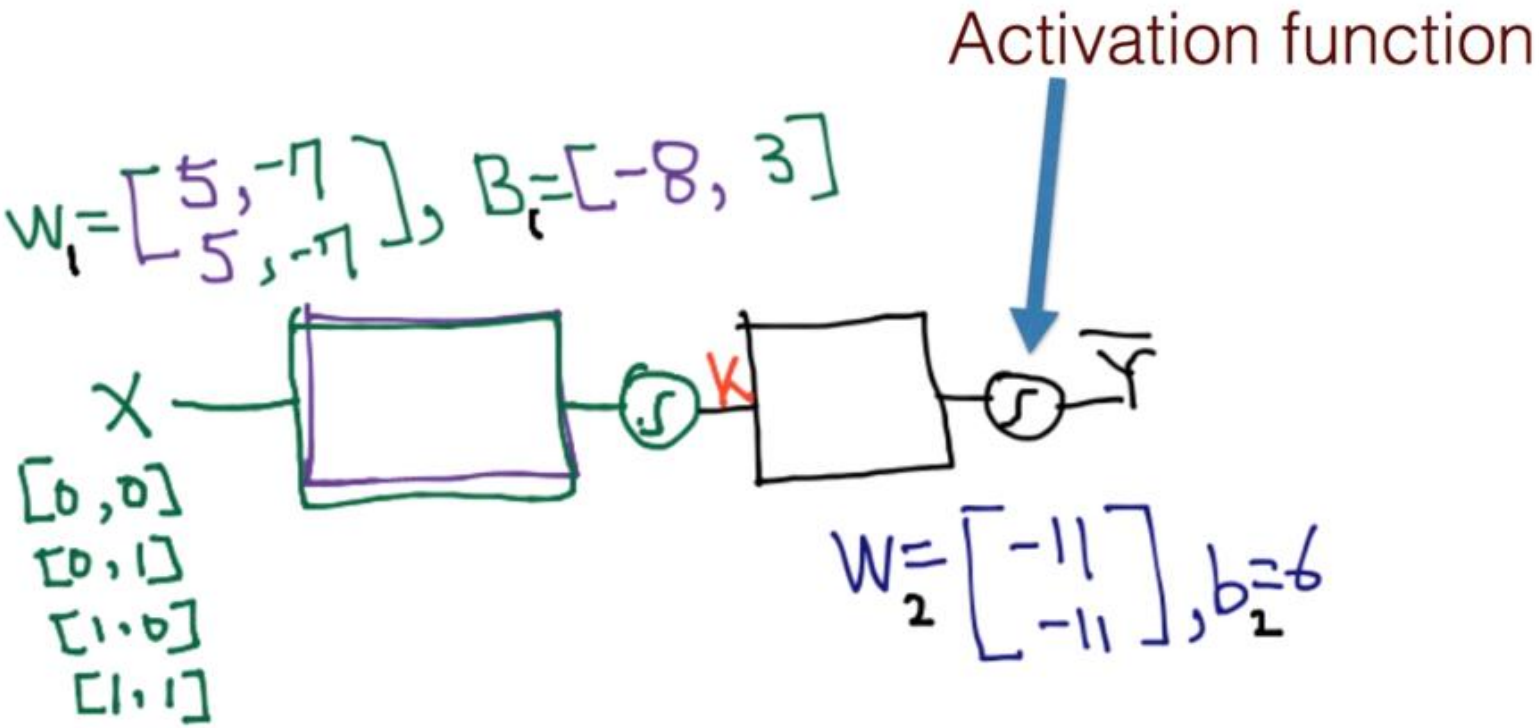


Back propagation in TensorFlow

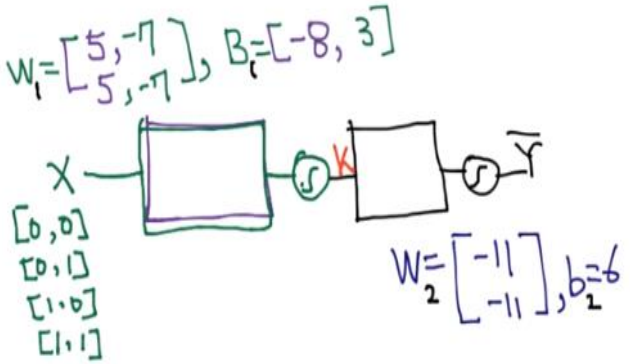
TensorBoard

```
# cost function
cost = -tf.reduce_mean(Y*tf.log(hypothesis) + (1-Y)*tf.log(1-hypothesis))
```

NN for XOR



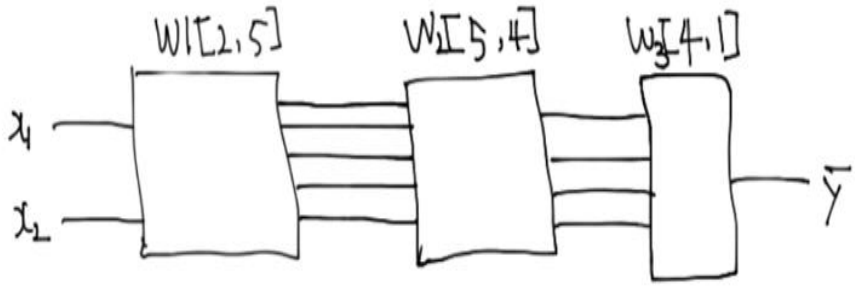
NN for XOR




```
W1 = tf.Variable(tf.random_uniform([2, 2], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([2, 1], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([2]), name="Bias1")
b2 = tf.Variable(tf.zeros([1]), name="Bias2")

# Our hypothesis
L2 = tf.sigmoid(tf.matmul(X, W1) + b1)
hypothesis = tf.sigmoid(tf.matmul(L2, W2) + b2)
```

Let's go deep and wide

```
W1 = tf.Variable(tf.random_uniform([2, 5], -1.0, 1.0))
W2 = tf.Variable(tf.random_uniform([5, 4], -1.0, 1.0))
W3 = tf.Variable(tf.random_uniform([4, 1], -1.0, 1.0))

b1 = tf.Variable(tf.zeros([5]), name="Bias1")
b2 = tf.Variable(tf.zeros([4]), name="Bias2")
b3 = tf.Variable(tf.zeros([1]), name="Bias2")

# Our hypothesis
L2 = tf.sigmoid(tf.matmul(X, W1) + b1)
L3 = tf.sigmoid(tf.matmul(L2, W2) + b2)
hypothesis = tf.sigmoid(tf.matmul(L3, W3) + b3)
```



9 Hidden Layers!

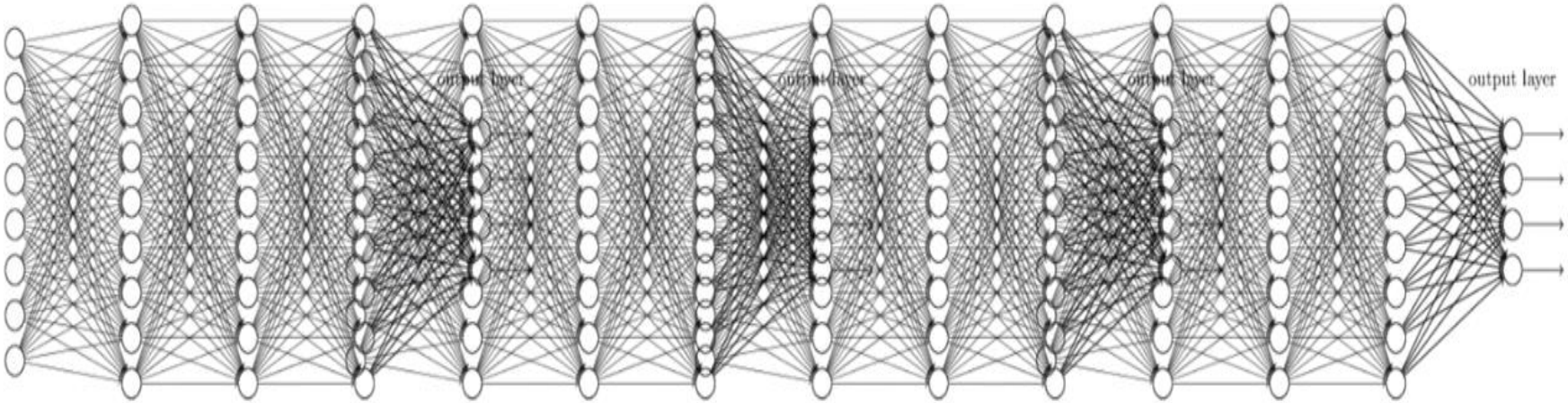
```
W1 = tf.Variable(tf.random_uniform([2, 5], -1.0, 1.0), name = "Weight1")

W2 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight2")
W3 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight3")
W4 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight4")
W5 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight5")
W6 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight6")
W7 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight7")
W8 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight8")
W9 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight9")
W10 = tf.Variable(tf.random_uniform([5, 5], -1.0, 1.0), name = "Weight10")

W11 = tf.Variable(tf.random_uniform([5, 1], -1.0, 1.0), name = "Weight11")

b1 = tf.Variable(tf.zeros([5]), name="Bias1")
b2 = tf.Variable(tf.zeros([5]), name="Bias2")
b3 = tf.Variable(tf.zeros([5]), name="Bias3")
b4 = tf.Variable(tf.zeros([5]), name="Bias4")
b5 = tf.Variable(tf.zeros([5]), name="Bias5")
b6 = tf.Variable(tf.zeros([5]), name="Bias6")
b7 = tf.Variable(tf.zeros([5]), name="Bias7")
b8 = tf.Variable(tf.zeros([5]), name="Bias8")
b9 = tf.Variable(tf.zeros([5]), name="Bias9")
b10 = tf.Variable(tf.zeros([5]), name="Bias10")

b11 = tf.Variable(tf.zeros([1]), name="Bias11")
```

```
# Our hypothesis
L1 = tf.sigmoid(tf.matmul(X, W1) + b1)
L2 = tf.sigmoid(tf.matmul(L1, W2) + b2)
L3 = tf.sigmoid(tf.matmul(L2, W3) + b3)
L4 = tf.sigmoid(tf.matmul(L3, W4) + b4)
L5 = tf.sigmoid(tf.matmul(L4, W5) + b5)
L6 = tf.sigmoid(tf.matmul(L5, W6) + b6)
L7 = tf.sigmoid(tf.matmul(L6, W7) + b7)
L8 = tf.sigmoid(tf.matmul(L7, W8) + b8)
L9 = tf.sigmoid(tf.matmul(L8, W9) + b9)
L10 = tf.sigmoid(tf.matmul(L9, W10) + b10)

hypothesis = tf.sigmoid(tf.matmul(L10, W11) + b11)
```

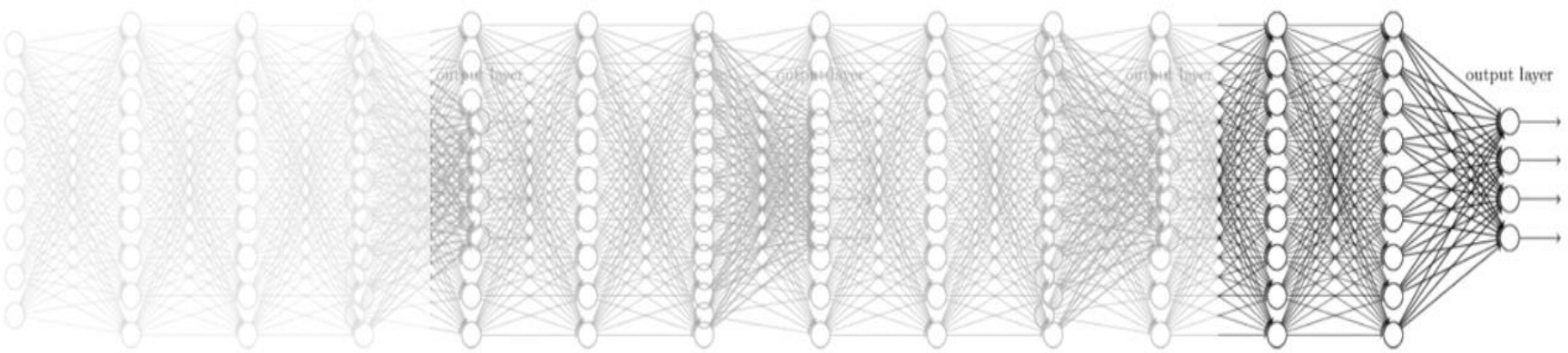
Poor Result

```
196000 [0.69314718, array([[ 0.49999988],
        [ 0.50000006],
        [ 0.49999982],
        [ 0.5       ]], dtype=float32)]
198000 [0.69314718, array([[ 0.49999988],
        [ 0.50000006],
        [ 0.49999982],
        [ 0.5       ]], dtype=float32)]
[array([[ 0.49999988],
        [ 0.50000006],
        [ 0.49999982],
        [ 0.5       ]], dtype=float32), array([[ 0.],
        [ 1.],
        [ 0.],
        [ 1.]], dtype=float32)]
Accuracy: 0.5
```

BP in Deep Learning



Vanishing Gradient (Winter2: 1986-2006)

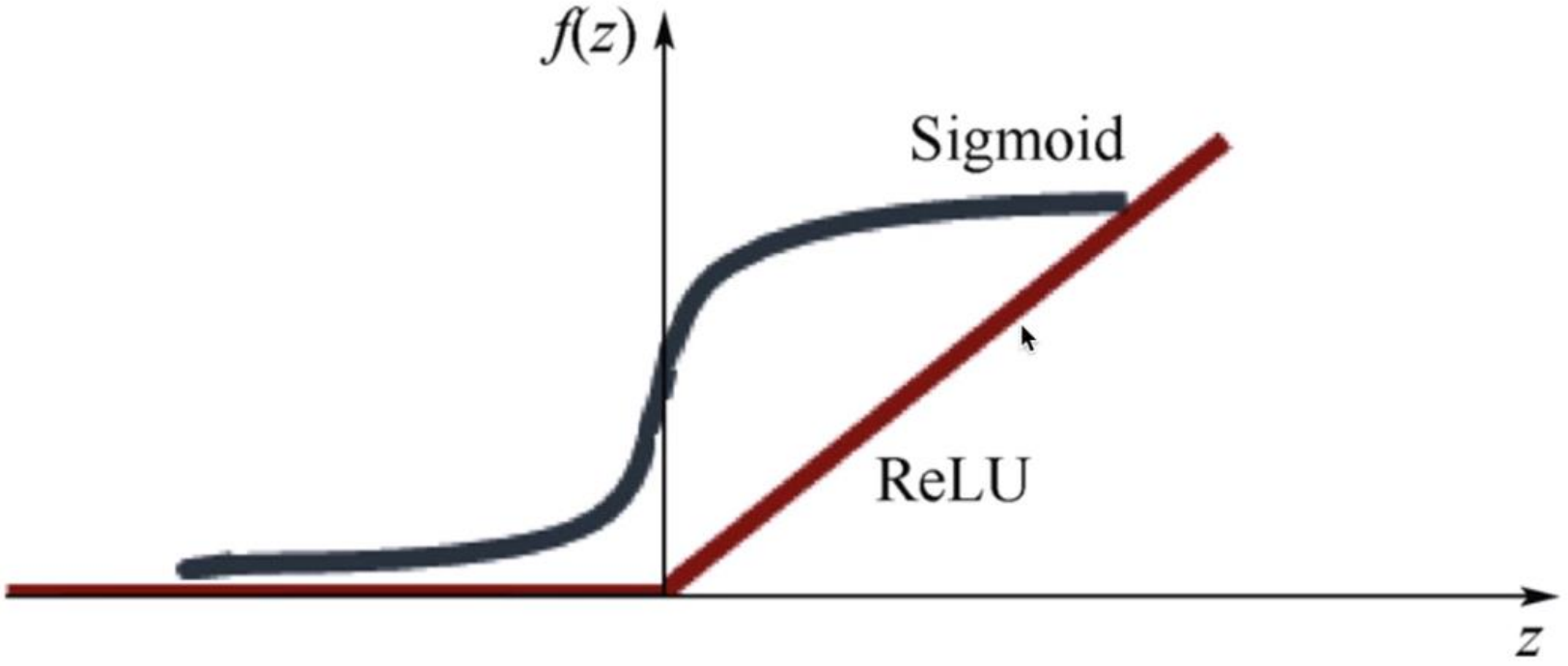


G. Hinton's Summary of finding

- Our labeled datasets were thousands of times too small.
- Our computers were millions of times too slow.
- We initialized the weights in a stupid way.
- We used the wrong type of non-linearity.

Sigmoid vs. ReLU

Rectified Linear Unit (ReLU)



ReLU

```
# Our hypothesis
with tf.name_scope("layer1") as scope:
    L1 = tf.nn.relu(tf.matmul(X, W1) + b1)
with tf.name_scope("layer2") as scope:
    L2 = tf.nn.relu(tf.matmul(L1, W2) + b2)
with tf.name_scope("layer3") as scope:
    L3 = tf.nn.relu(tf.matmul(L2, W3) + b3)
with tf.name_scope("layer4") as scope:
    L4 = tf.nn.relu(tf.matmul(L3, W4) + b4)
with tf.name_scope("layer5") as scope:
    L5 = tf.nn.relu(tf.matmul(L4, W5) + b5)
with tf.name_scope("layer6") as scope:
    L6 = tf.nn.relu(tf.matmul(L5, W6) + b6)
with tf.name_scope("layer7") as scope:
    L7 = tf.nn.relu(tf.matmul(L6, W7) + b7)
with tf.name_scope("layer8") as scope:
    L8 = tf.nn.relu(tf.matmul(L7, W8) + b8)
with tf.name_scope("layer9") as scope:
    L9 = tf.nn.relu(tf.matmul(L8, W9) + b9)
with tf.name_scope("layer10") as scope:
    L10 = tf.nn.relu(tf.matmul(L9, W10) + b10)

with tf.name_scope("last") as scope:
    hypothesis = tf.sigmoid(tf.matmul(L10, W11) + b11)
```

Work Very Well!

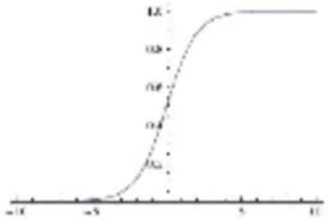
```
196000 [2.6226094e-06, array([[  2.59195826e-06],
        [  9.99999642e-01],
        [  9.99994874e-01],
        [  2.43454133e-06]], dtype=float32)]
198000 [2.607708e-06, array([[  2.55822852e-06],
        [  9.99999642e-01],
        [  9.99994874e-01],
        [  2.40260101e-06]], dtype=float32)]
[array([[  2.52509381e-06],
        [  9.99999642e-01],
        [  9.99994874e-01],
        [  2.37124709e-06]], dtype=float32), array([[ 0.],
        [ 1.],
        [ 1.],
        [ 0.]], dtype=float32)]
Accuracy: 1.0
```

Other Activation Functions

Activation Functions

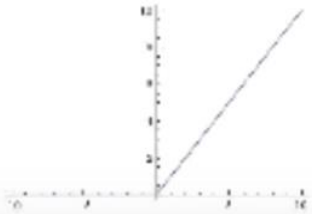
Sigmoid

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

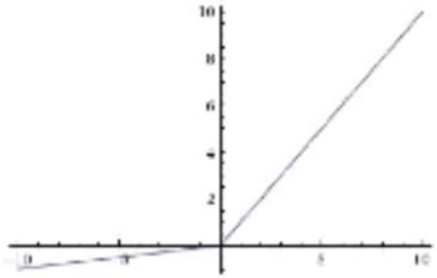


tanh tanh(x)

ReLU max(0,x)

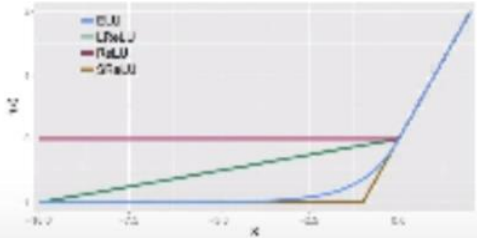


Leaky ReLU
max(0.1x, x)



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

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



Activation Functions Evaluation

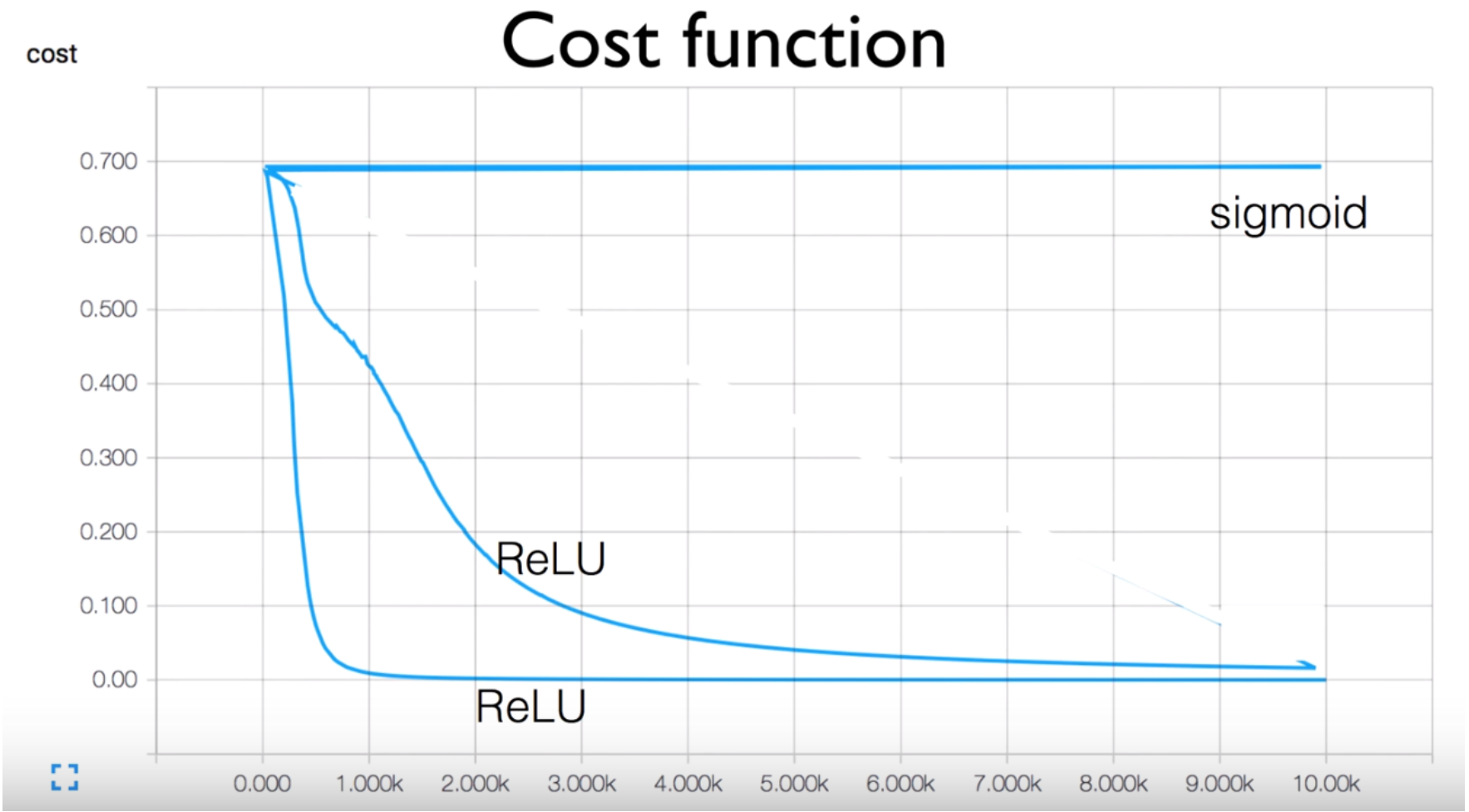
maxout	ReLU	VLReLU	tanh	Sigmoid
93.94	**92.11**	92.97	89.28	n/c
93.78	91.74	92.40	89.48	n/c
–	91.93	**93.09**	–	n/c
91.75	90.63	92.27	**89.82**	n/c
n/c†	90.91	92.43	89.54	n/c

[Mishkin et al. 2015]

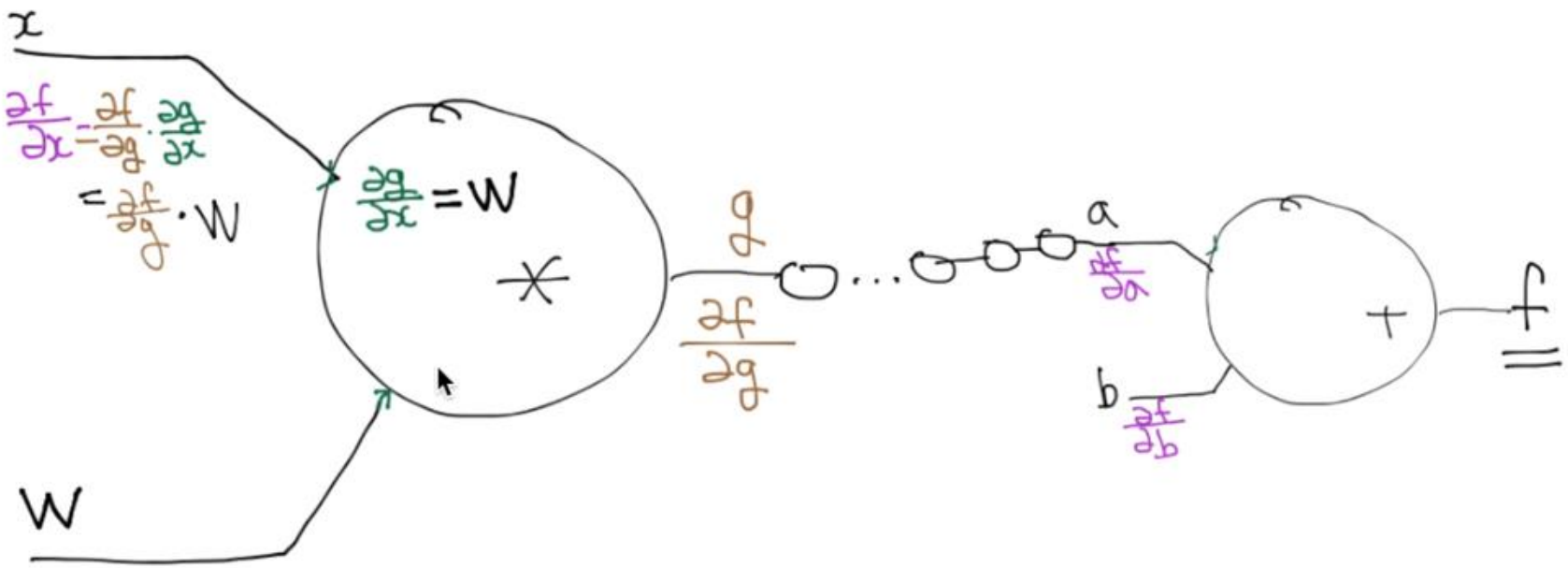
G. Hinton's Summary of finding

- Our labeled datasets were thousands of times too small.
- Our computers were millions of times too slow.
- We initialized the weights in a stupid way.
- We used the wrong type of non-linearity.

Cost Function



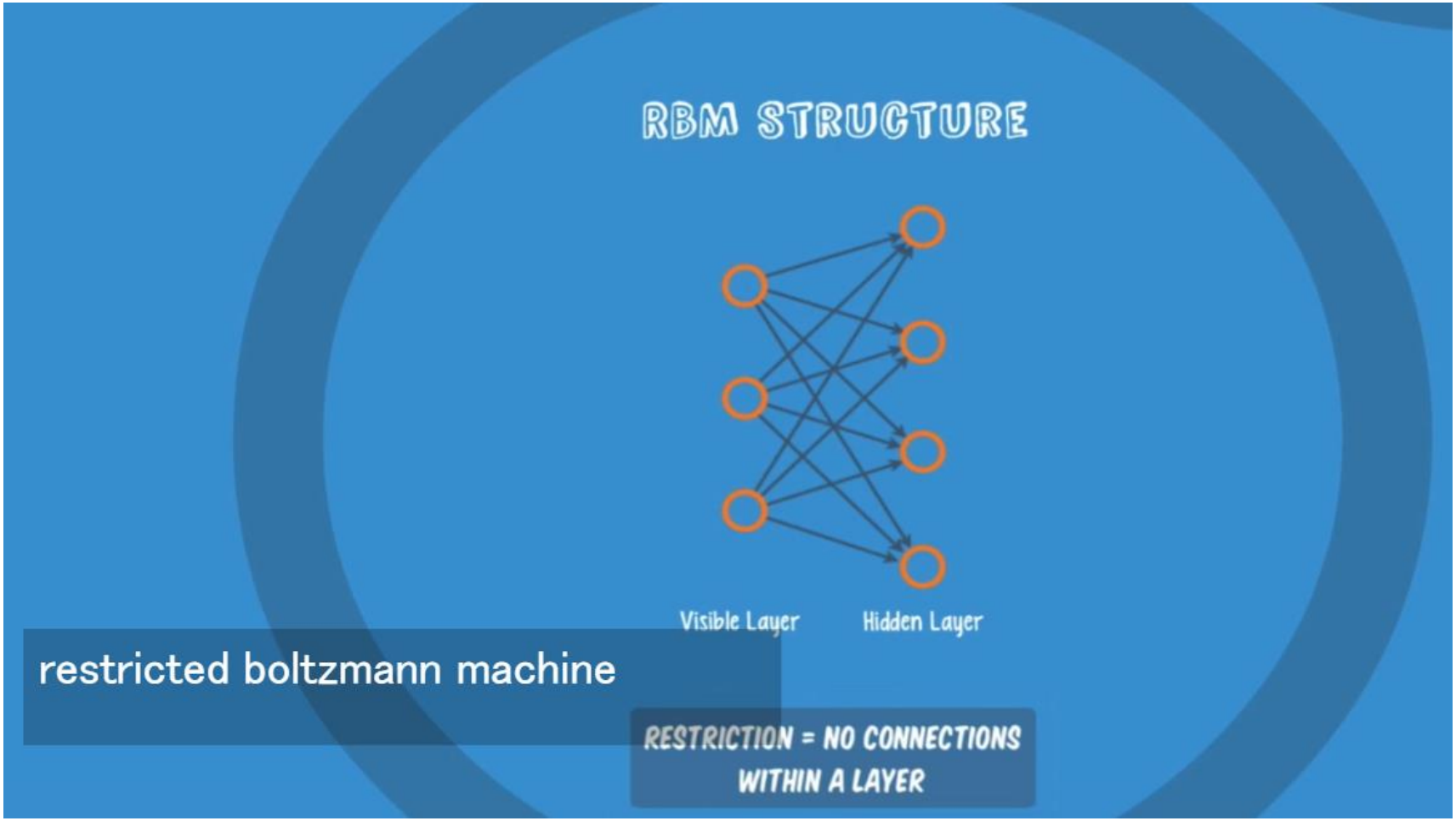
Set all initial weights to 0



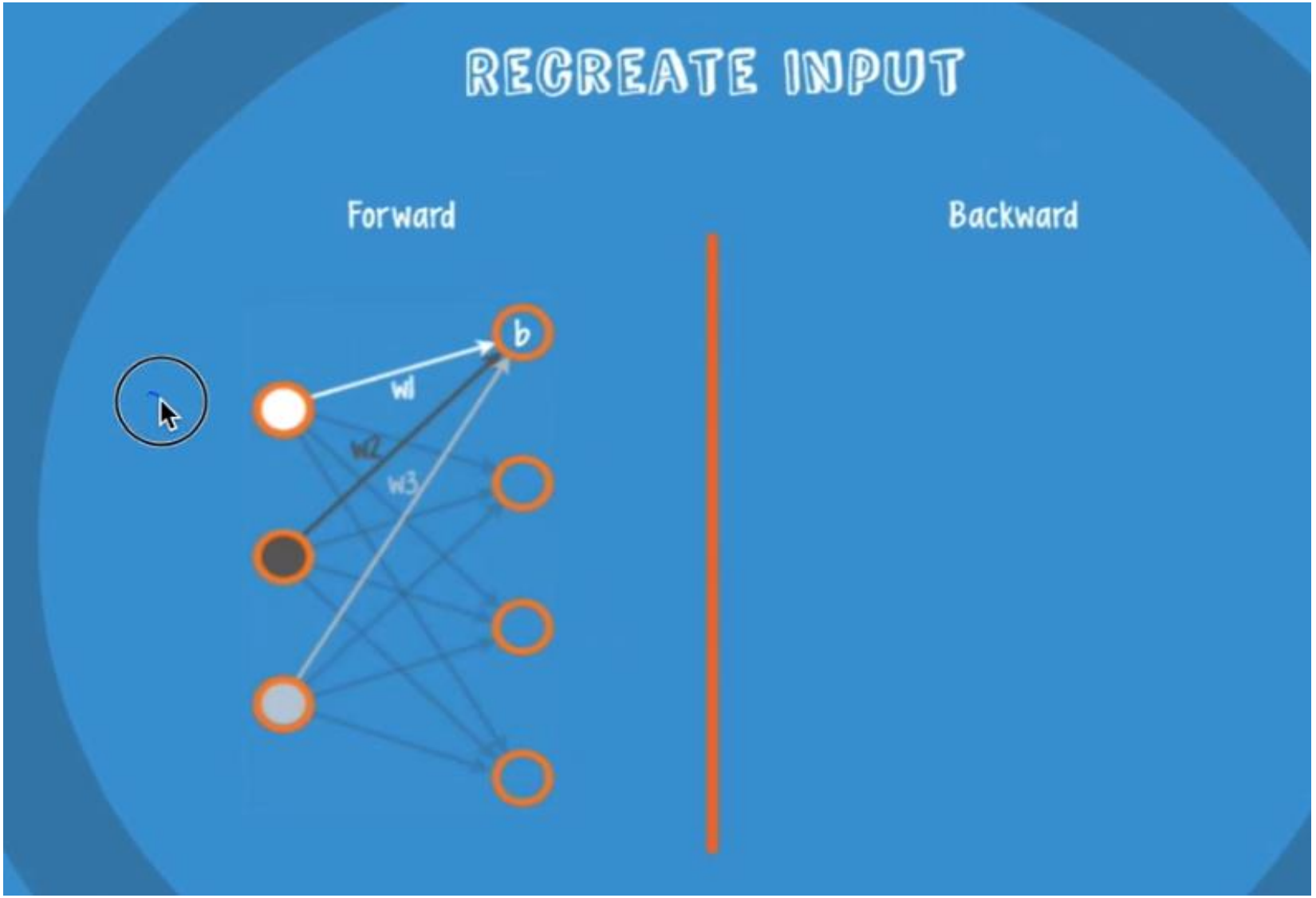
Set the initial weight values wisely!

- Not all 0's
- Challenging issue
- Hinton et al. (2006) "A Fast Learning Algorithm for Deep Belief Nets"
  - Restricted Boltzmann Machine

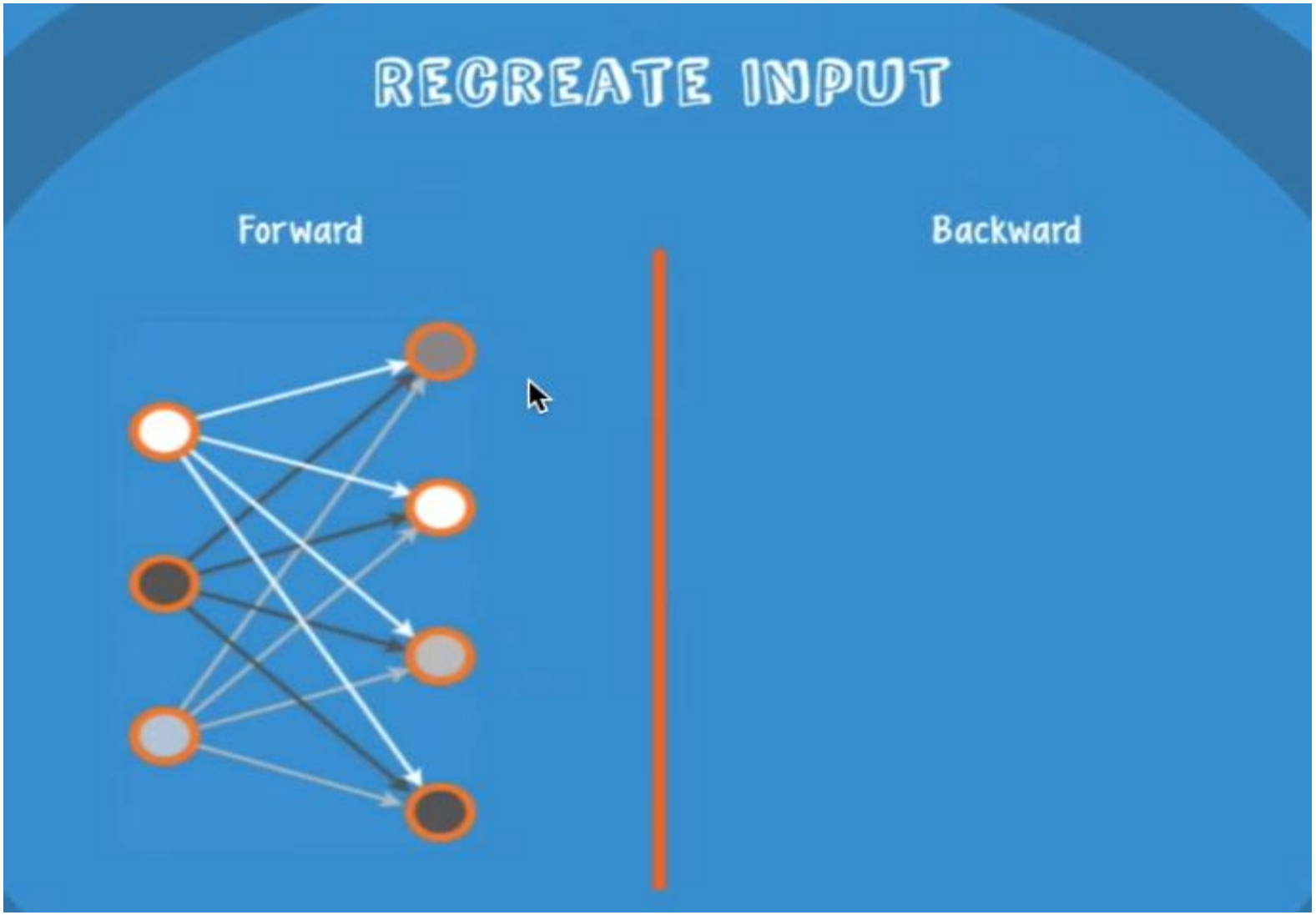
RBM


RBM STRUCTURE
Visible Layer
Hidden Layer
restricted boltzmann machine
RESTRICTION = NO CONNECTIONS
WITHIN A LAYER

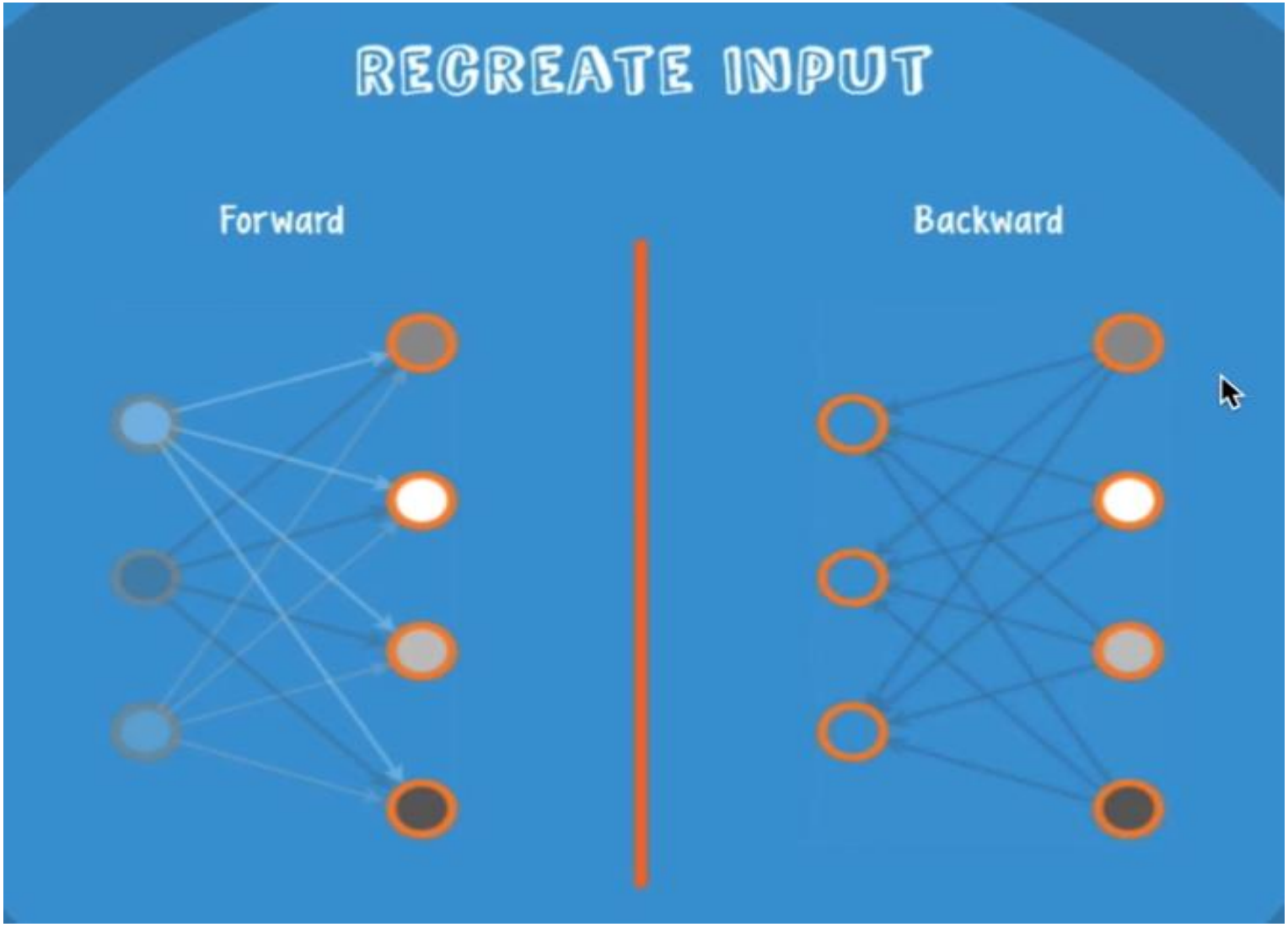
RBM



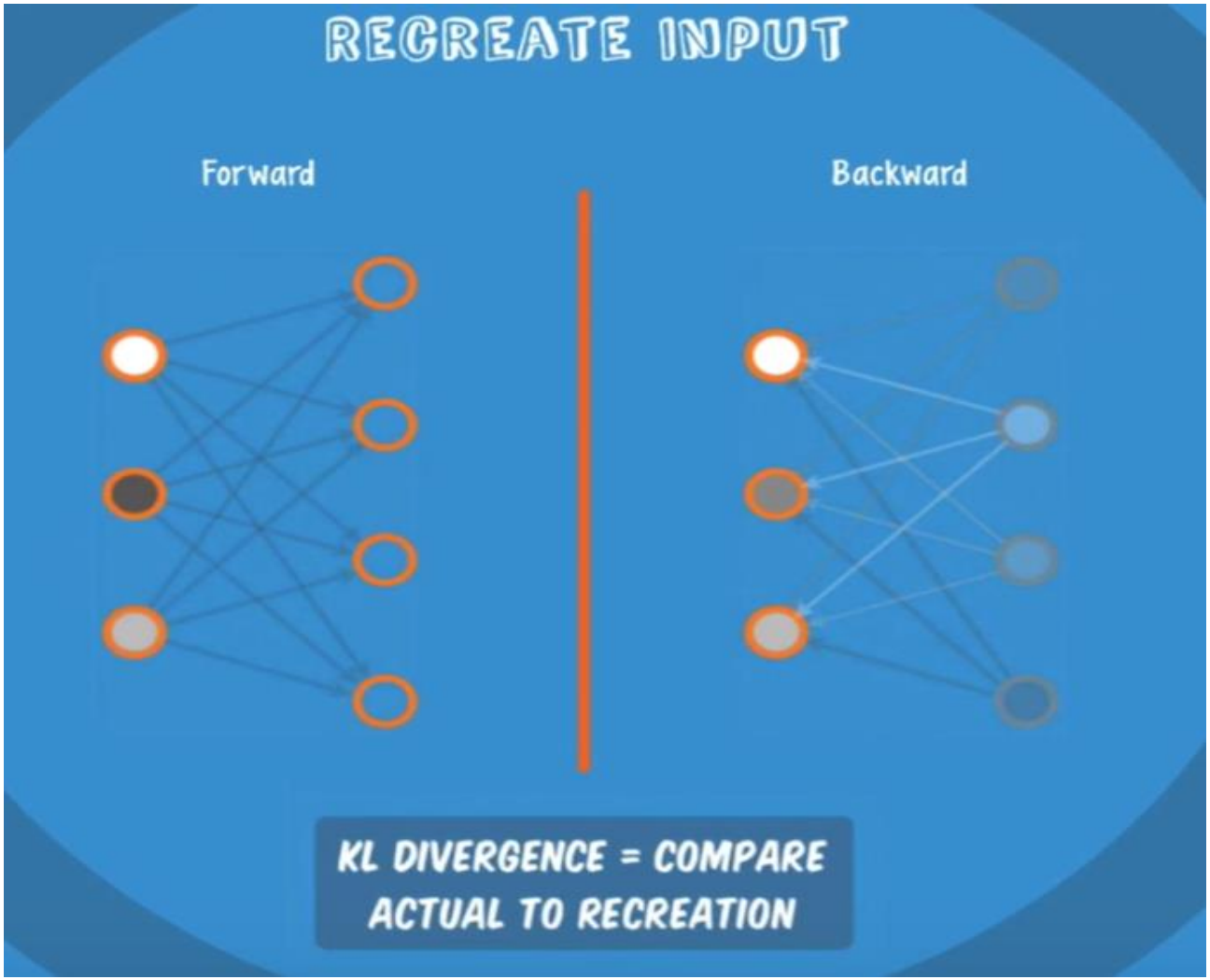
RBM



RBM



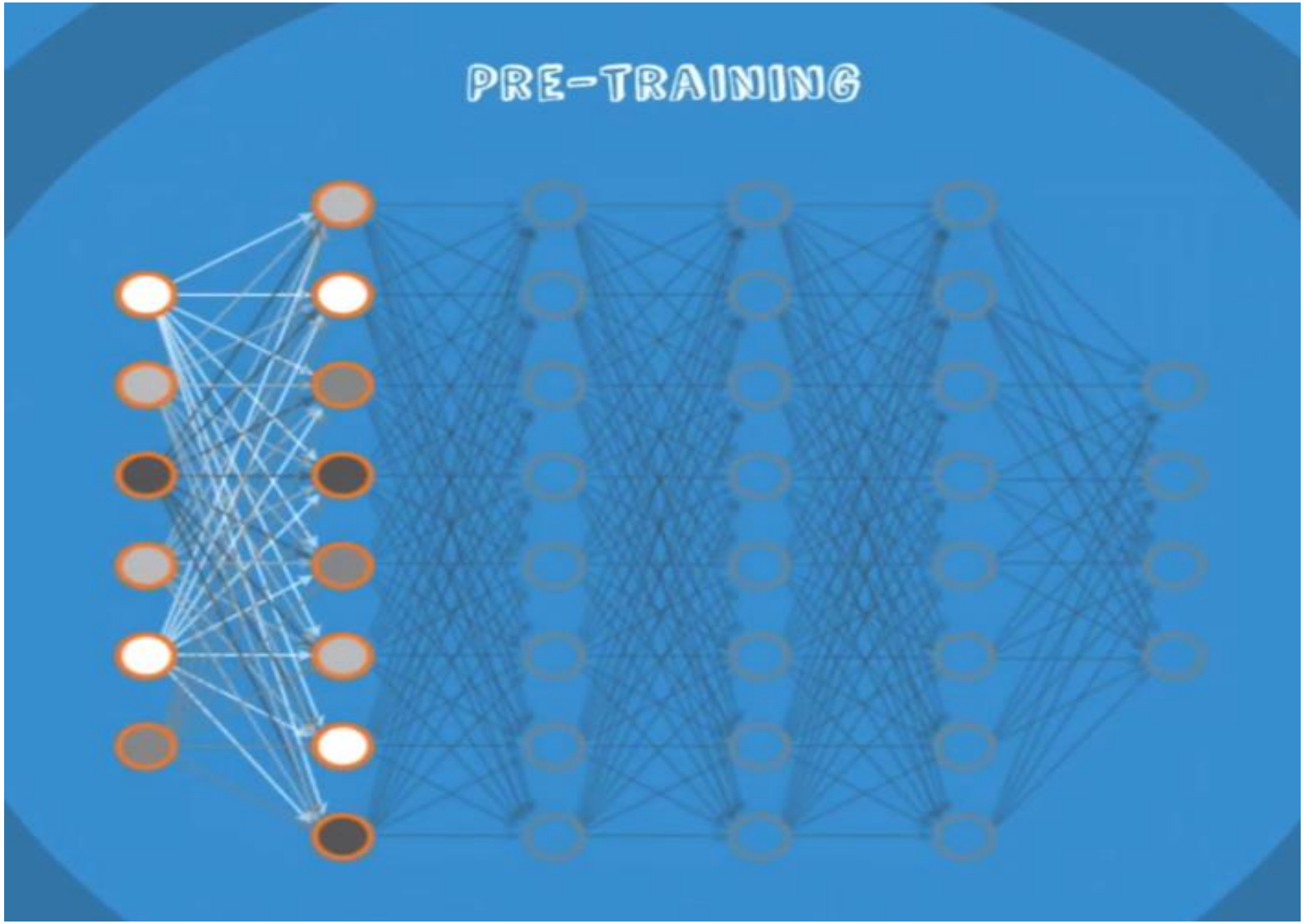
RBM


RECREATE INPUT
Forward
Backward
KL DIVERGENCE = COMPARE
ACTUAL TO RECREATION

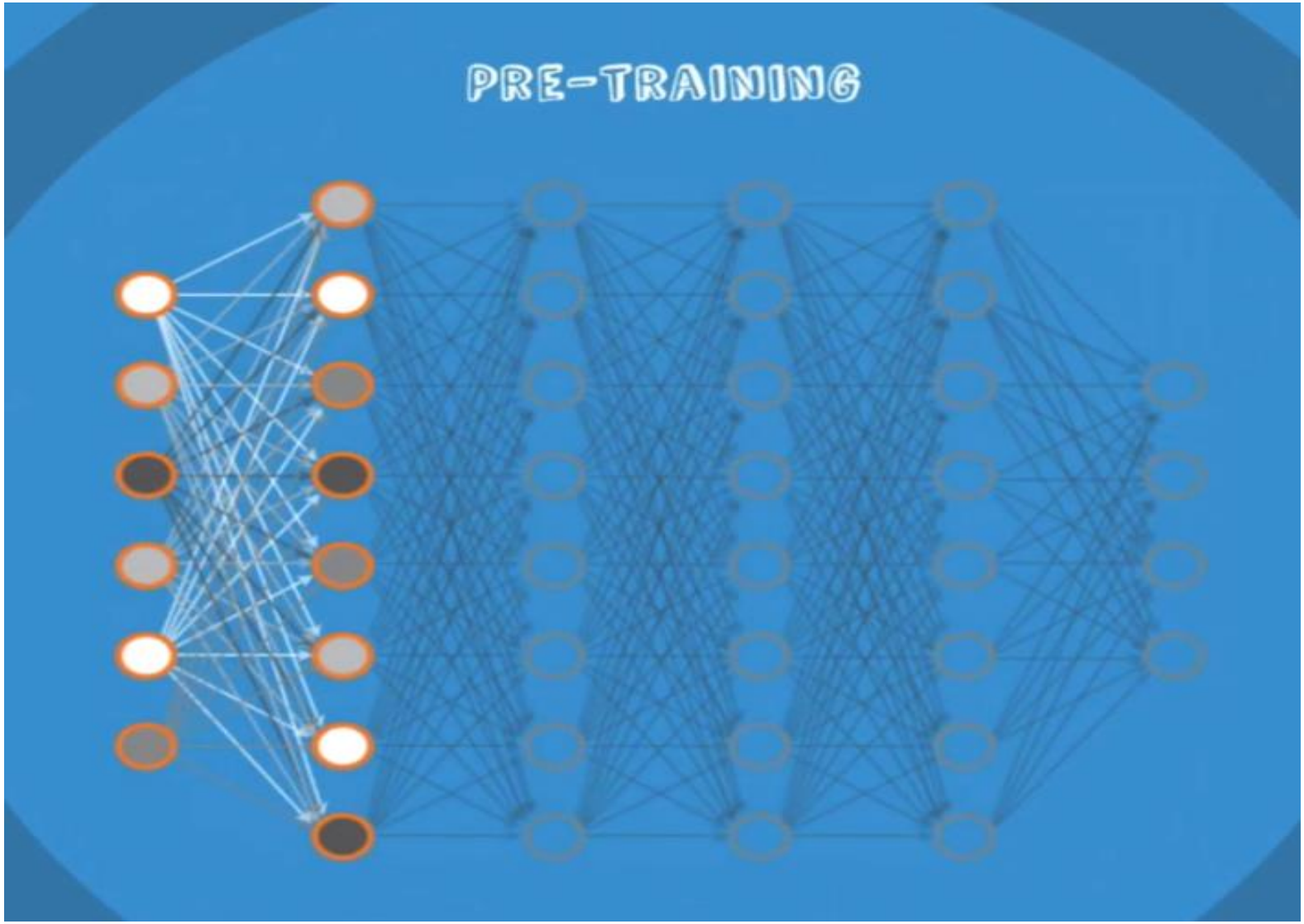
How to use RBM to initialize weight

- Apply the RBM idea on adjacent two layers as a pre-training step
- Continue the first process to all layers
- This will set weights
- Example: Deep Belief Network
  - Weight initialized by RBM

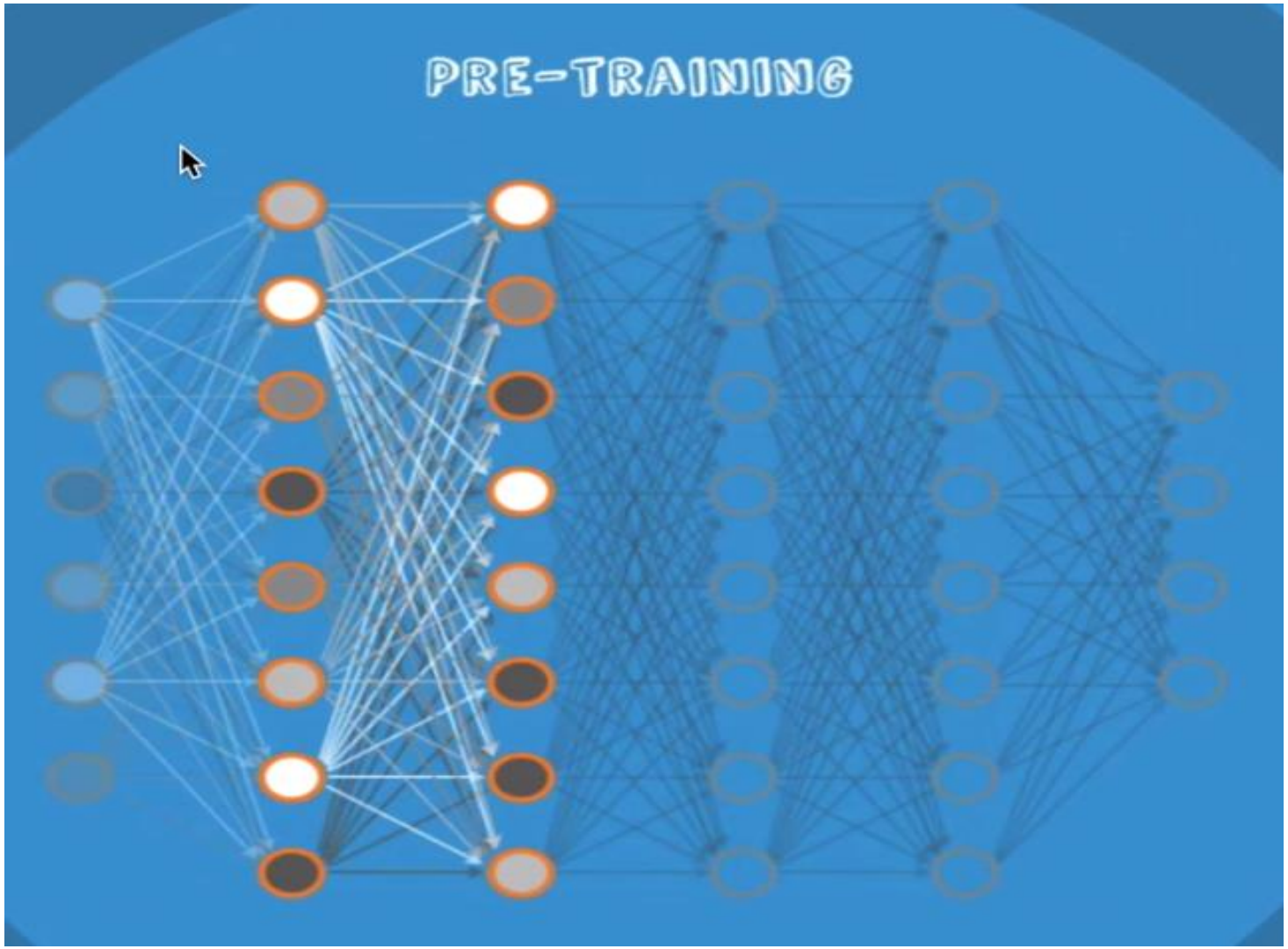
Applying RBM



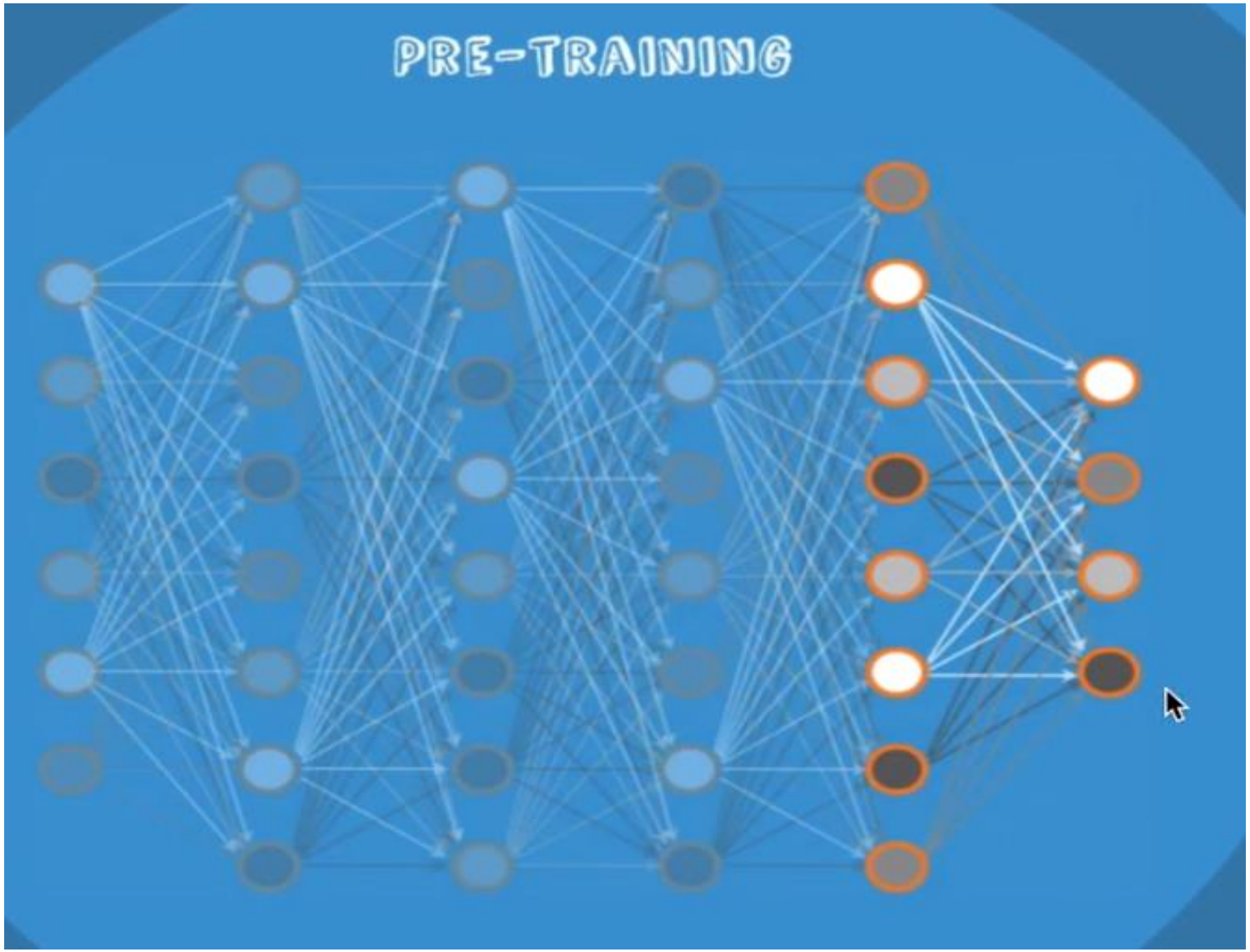
Applying RBM



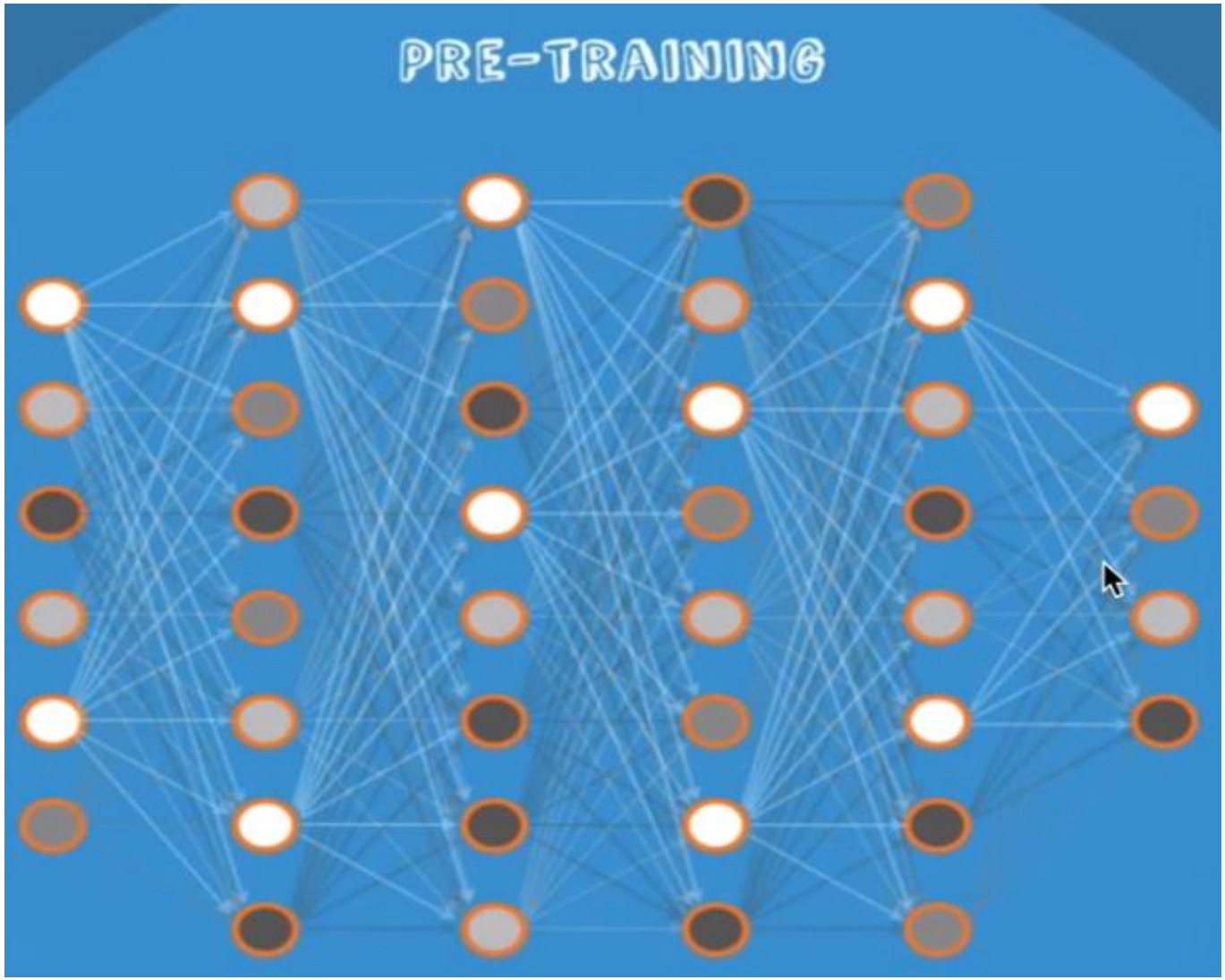
Applying RBM



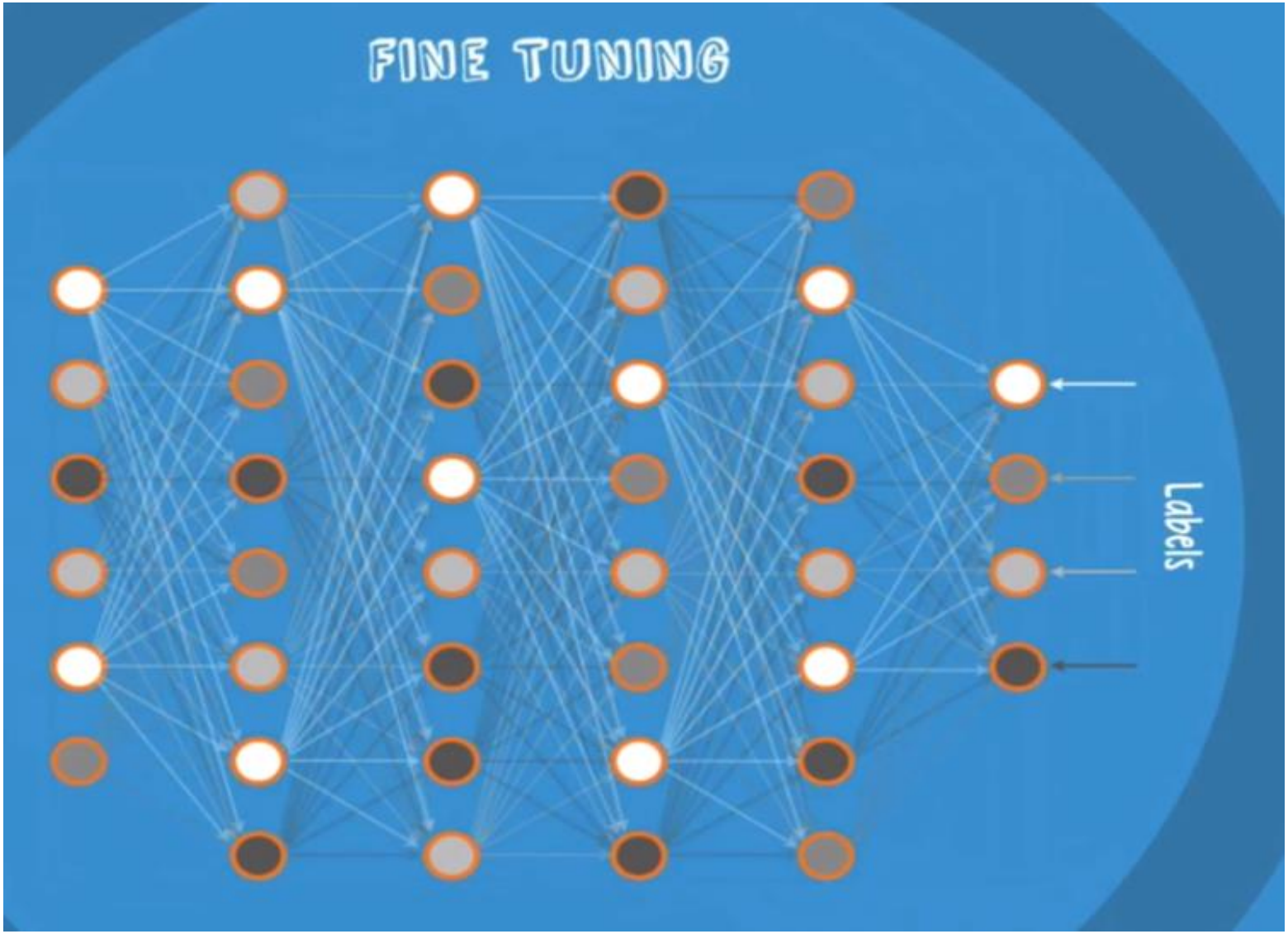
Applying RBM



Applying RBM



Real Training: Fine Tuning



Good News

- No need to use complicated RBM for weight initializations
- Simple methods are OK
  - **Xavier initialization**: X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in International conference on artificial intelligence and statistics, 2010
  - **He's initialization**: K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification," 2015

Xavir/He Initialization

- Makes sure the weights are 'just right', not too small, not too big
- Using number of input (fan_in) and output (fan_out)

```
# Xavier initialization
# Glorot et al. 2010
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in)

# He et al. 2015
W = np.random.randn(fan_in, fan_out)/np.sqrt(fan_in/2)
```

Activation Functions and Initialization

Init method	maxout	ReLU	VLReLU	tanh	Sigmoid
LSUV	**93.94**	**92.11**	92.97	89.28	n/c
OrthoNorm	93.78	91.74	92.40	89.48	n/c
OrthoNorm-MSRA scaled	–	91.93	**93.09**	–	n/c
Xavier	91.75	90.63	92.27	**89.82**	n/c
MSRA	n/c†	90.91	92.43	89.54	n/c

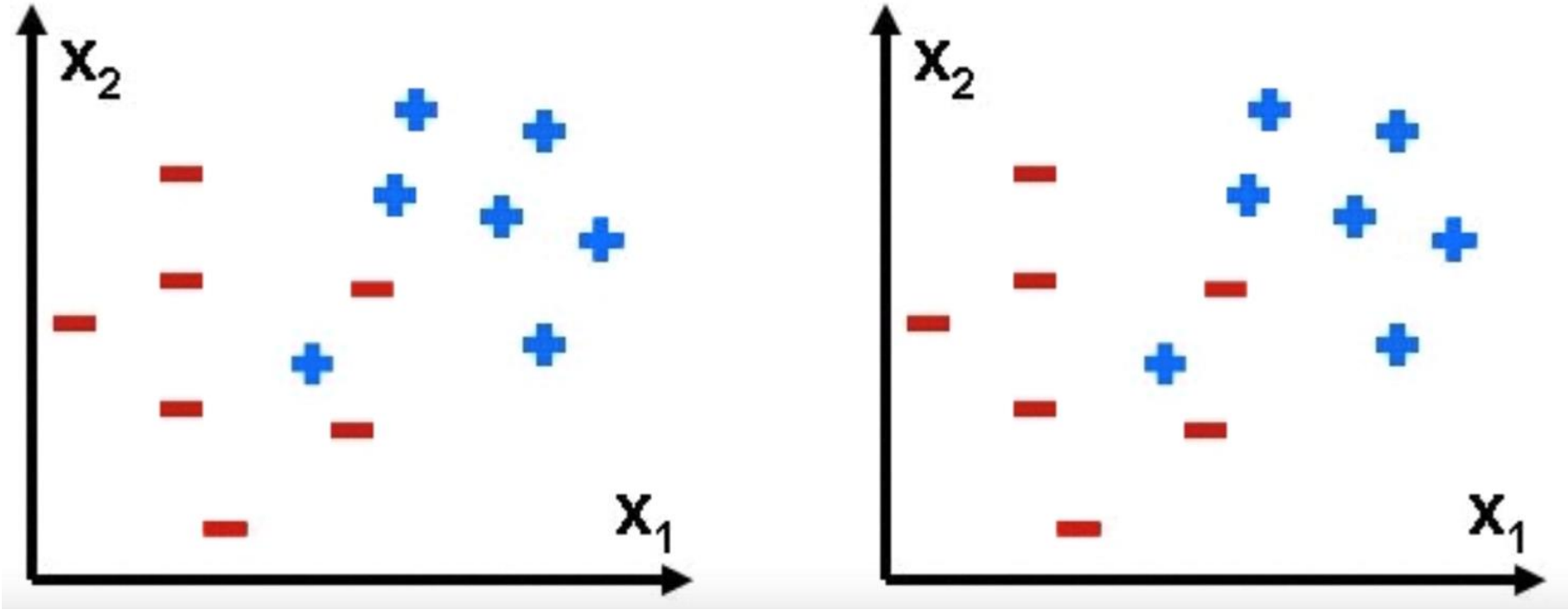
[Mishkin et al. 2015]

Hinton's Summary

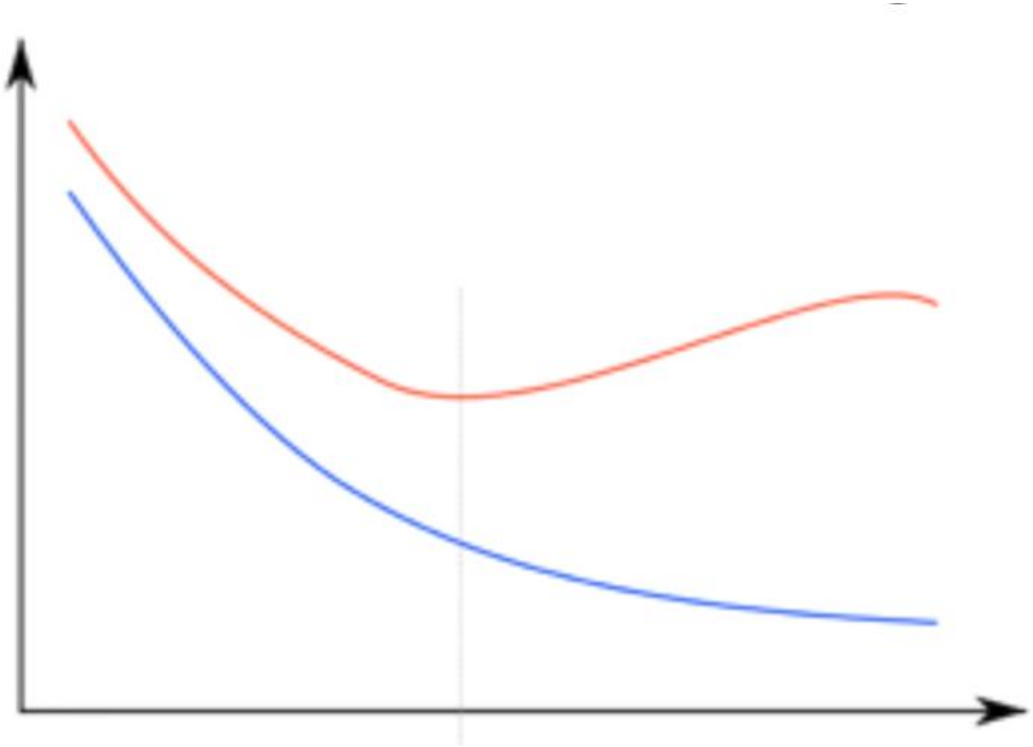
- Our labeled datasets were thousands of times too small.
- Our computers were millions of times too slow.
- **We initialized the weights in a stupid way.**
- **We used the wrong type of non-linearity.**

Dropout

Overfitting



Am I Overfitting?



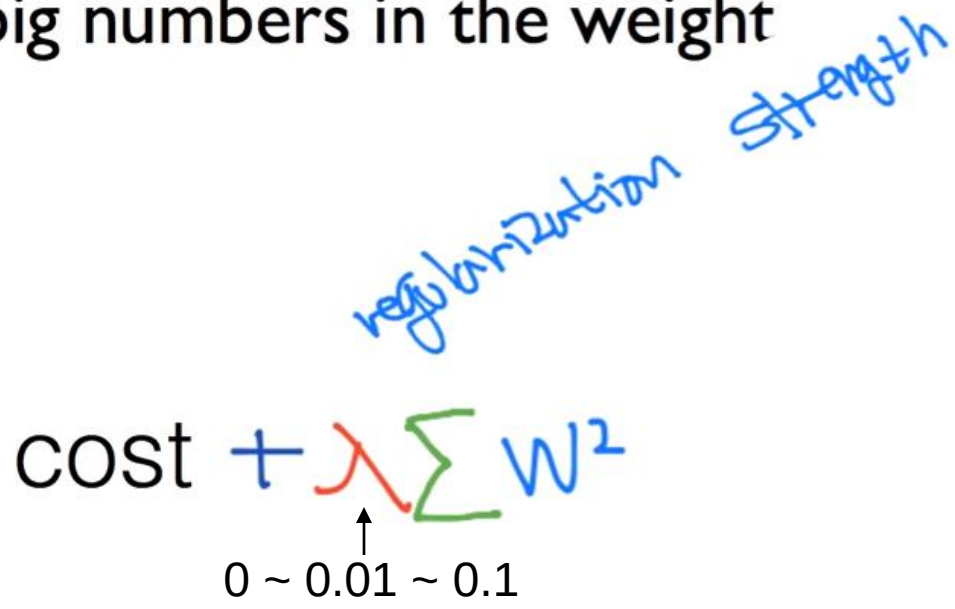
- Very high accuracy on the training dataset (eg: 0.99)
- Poor accuracy on the test data set (0.85)

Solutions for Overfitting

- More training data!
- Reduce the number of features
- **Regularization**

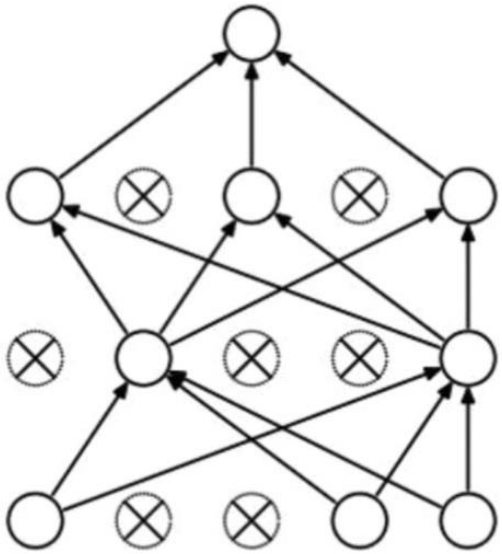
Regularization

- Let's not have too big numbers in the weight




```
l2reg = 0.001 * tf.reduce_sum(tf.square(W))
```

How could this be a good idea?



Forces the network to have a redundant representation.

TensorFlow Implementation

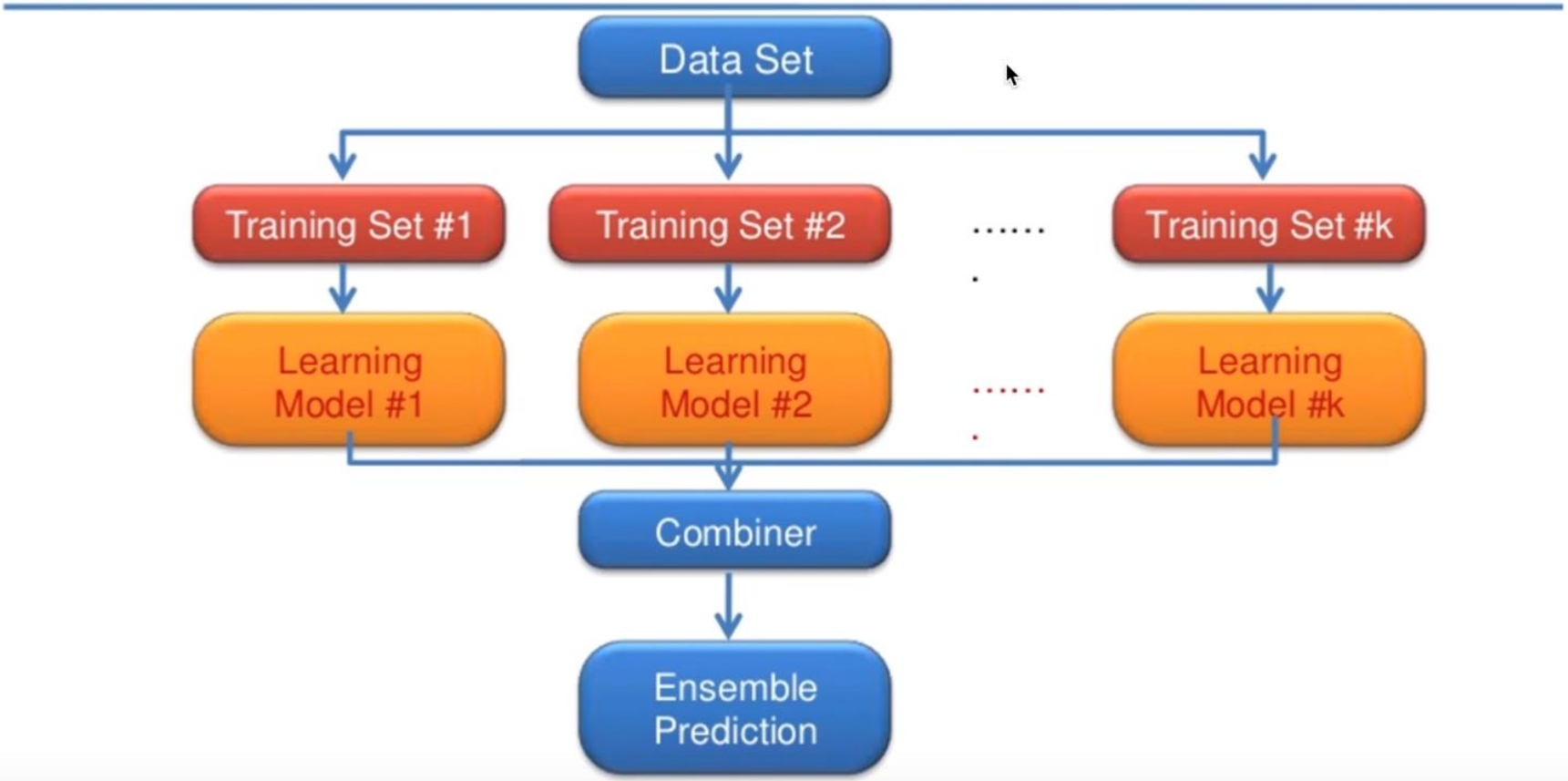
```
dropout_rate = tf.placeholder("float")
_L1 = tf.nn.relu(tf.add(tf.matmul(X, W1), B1))
L1 = tf.nn.dropout(_L1, dropout_rate)
```

```
TRAIN:
  sess.run(optimizer, feed_dict={X: batch_xs, Y: batch_ys,
  dropout_rate: 0.7})
```

```
EVALUATION:
  print "Accuracy:", accuracy.eval({X: mnist.test.images, Y:
  mnist.test.labels, dropout_rate: 1})
```

What is Ensemble?

Accuracy Improvement: 2 ~ 5%



Convolutional Neural Network

CNN History

A bit of history:

Hubel & Wiesel,

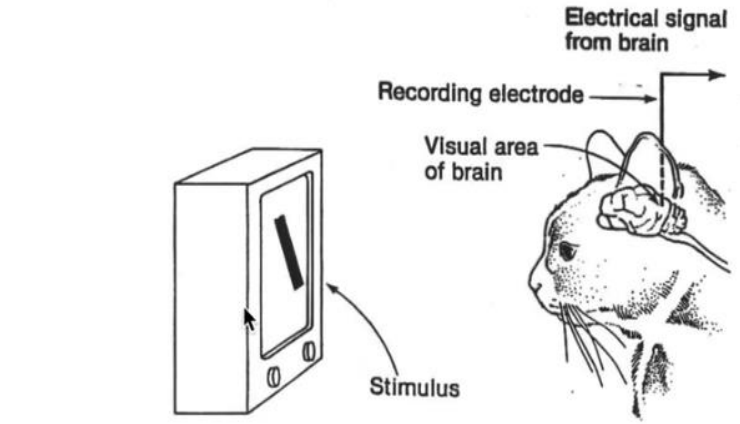
1959

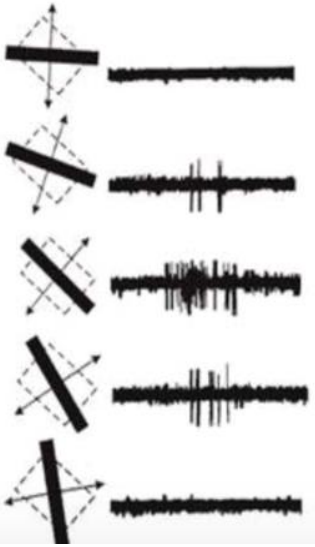
RECEPTIVE FIELDS OF SINGLE NEURONES IN
THE CAT'S STRIATE CORTEX

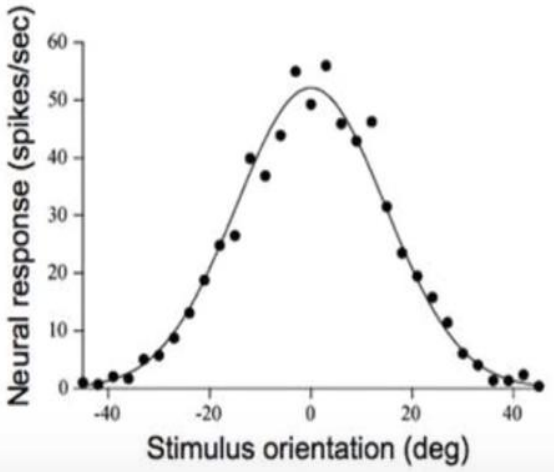
1962

RECEPTIVE FIELDS, BINOCULAR INTERACTION
AND FUNCTIONAL ARCHITECTURE IN
THE CAT'S VISUAL CORTEX

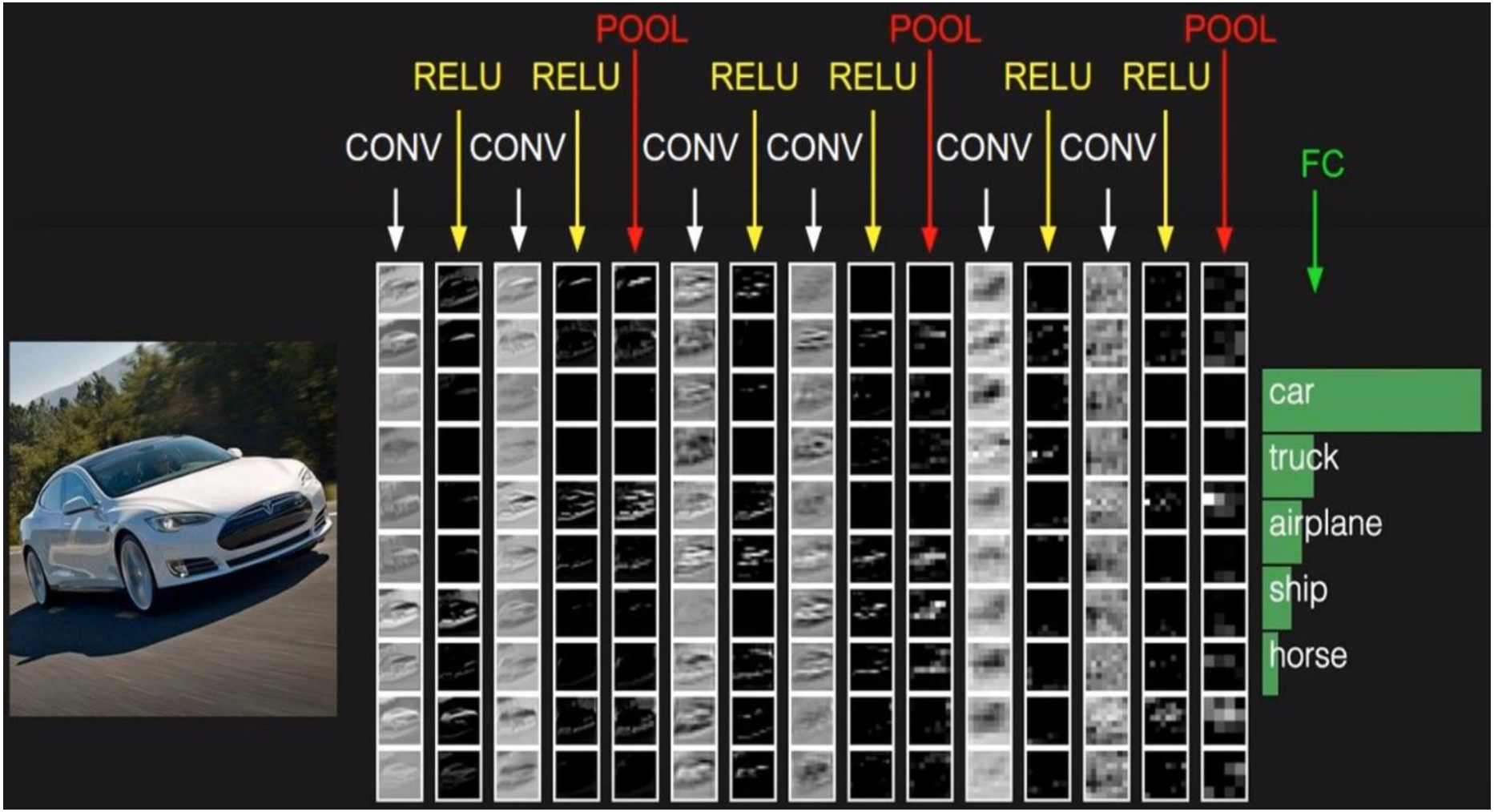
1968...



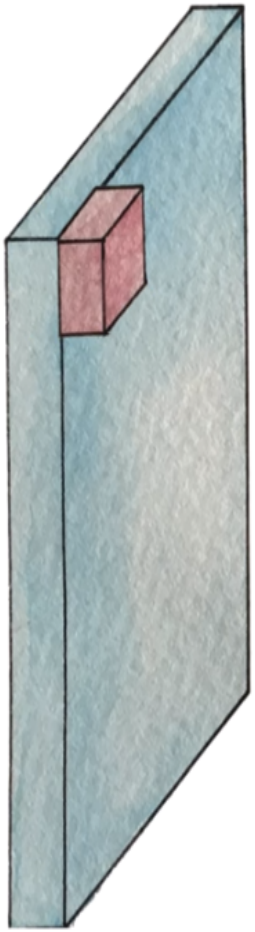




Layers by Convolution



Let's focus on a small area only (5x5x3)

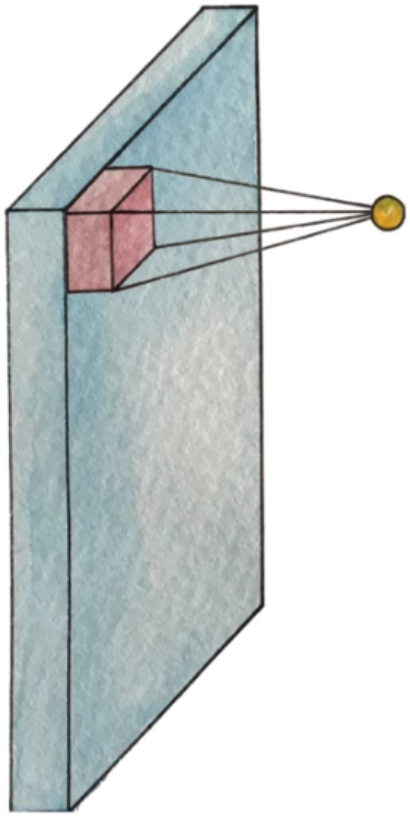


32x32x3 image

5x5x3 filter

Get One number using the filter

Get one number using the filter

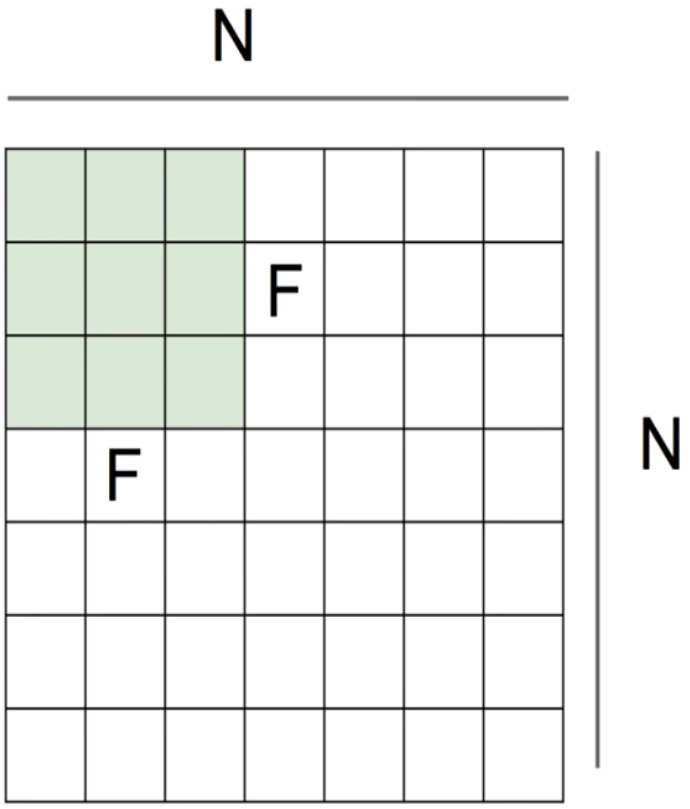


one number! =Wx+b =ReLU(Wx+b)

5x5x3 filter

32x32x3 image

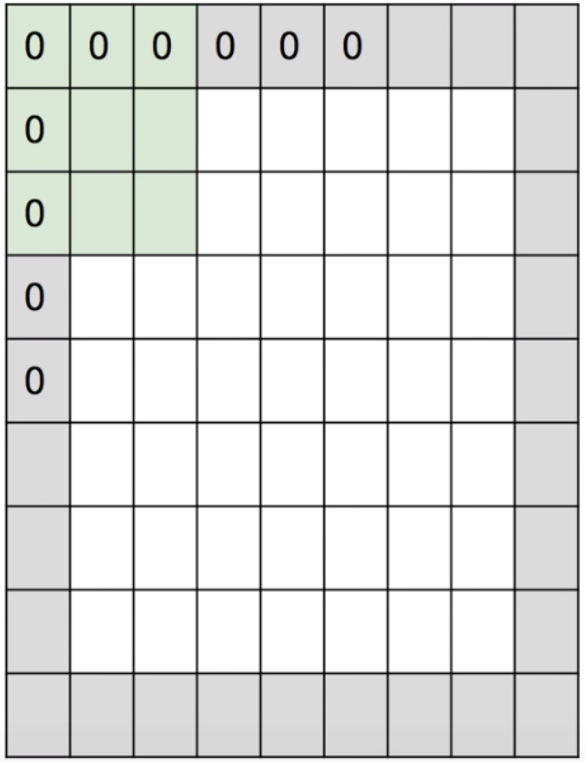
Output size after convolution



Output size:
(N - F) / stride + 1

e.g. 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 :\

Common to zero pad the border



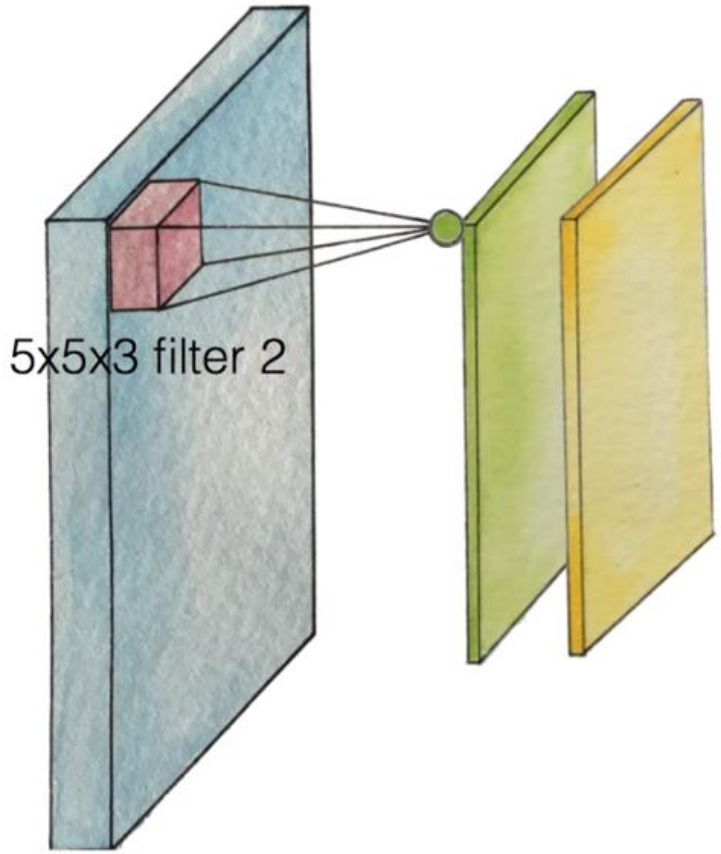
e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7X7 Output!

(recall:)
(N - F) / stride + 1

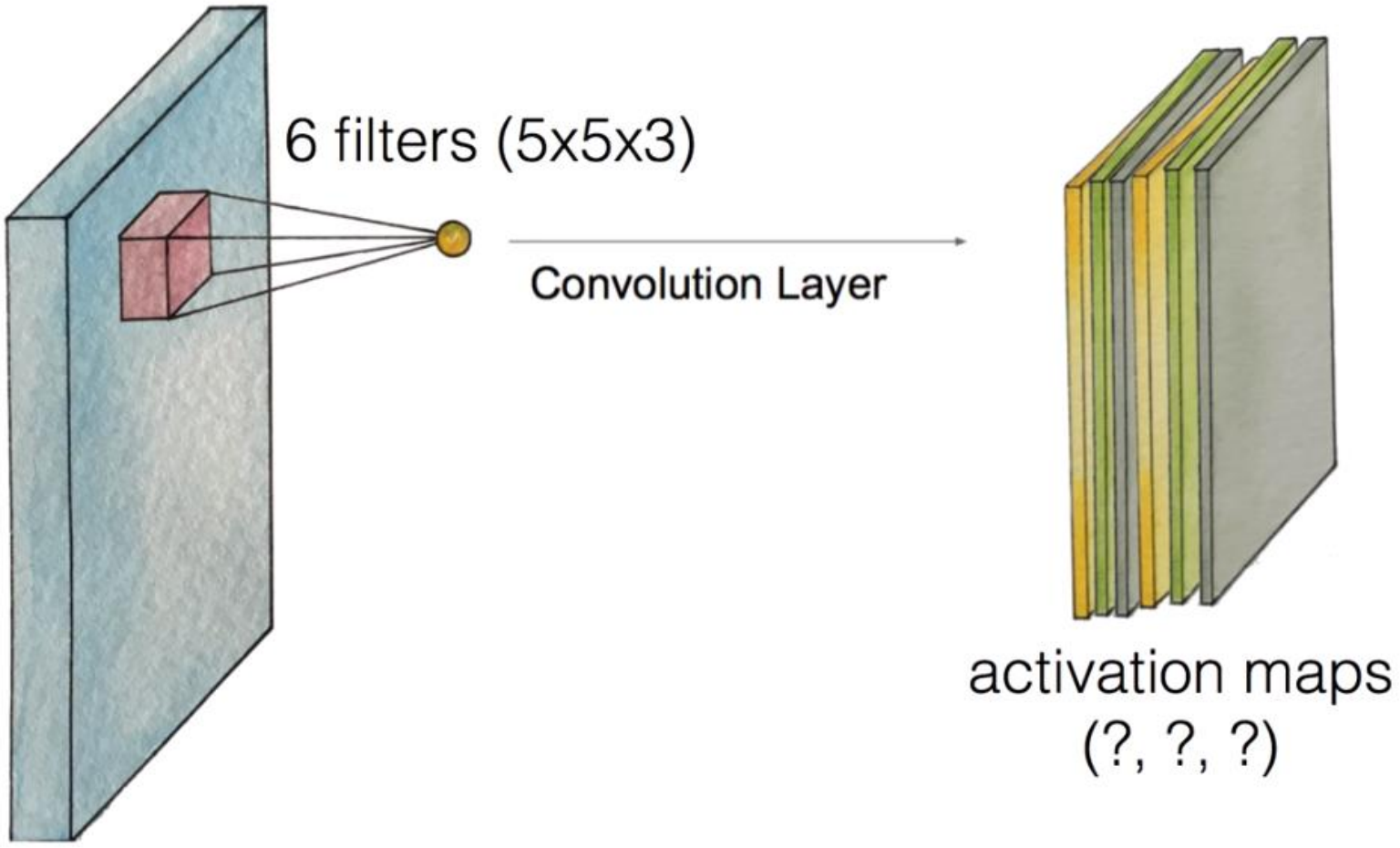
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Swiping the entire image

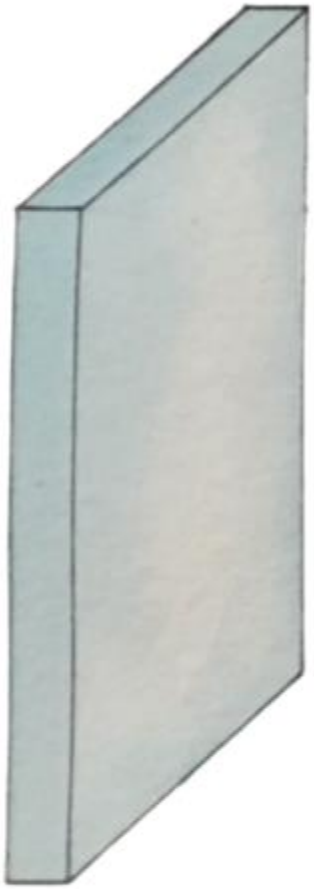


32x32x3 image

Swiping the entire image



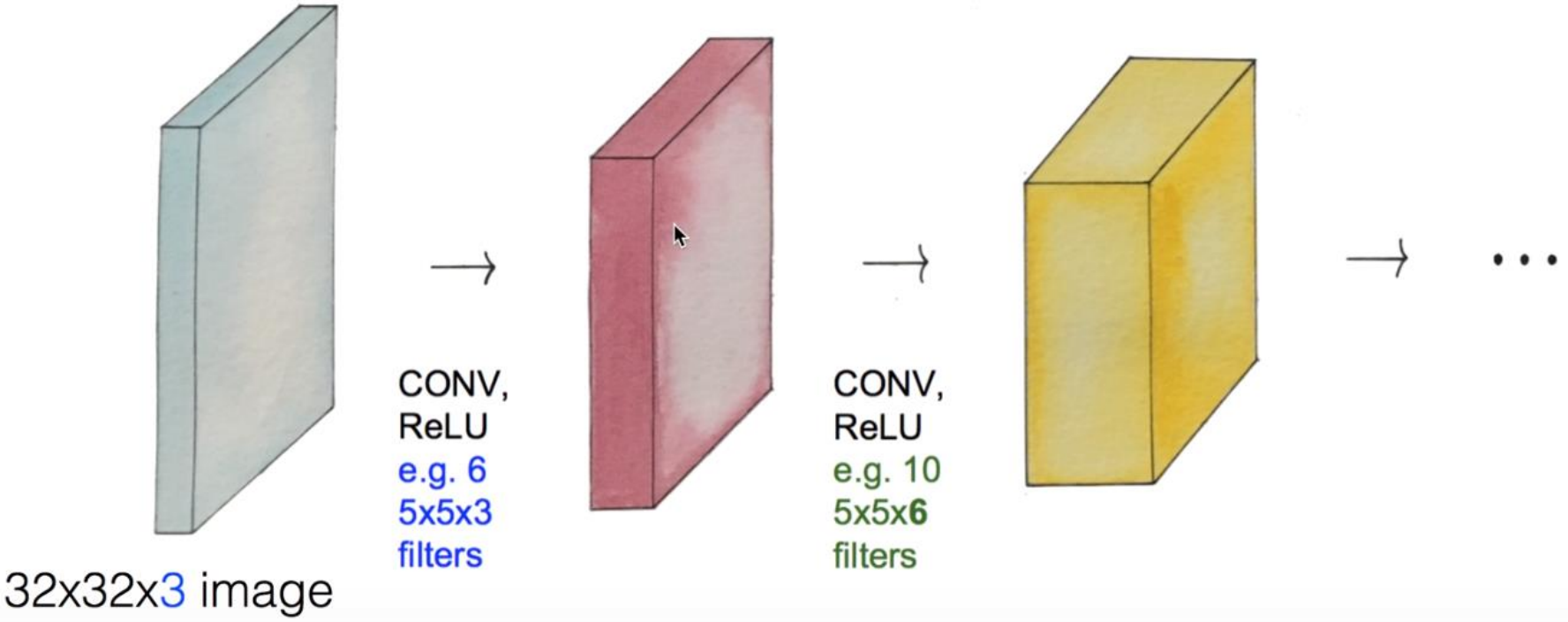
Convolution Layers



32x32x3 image

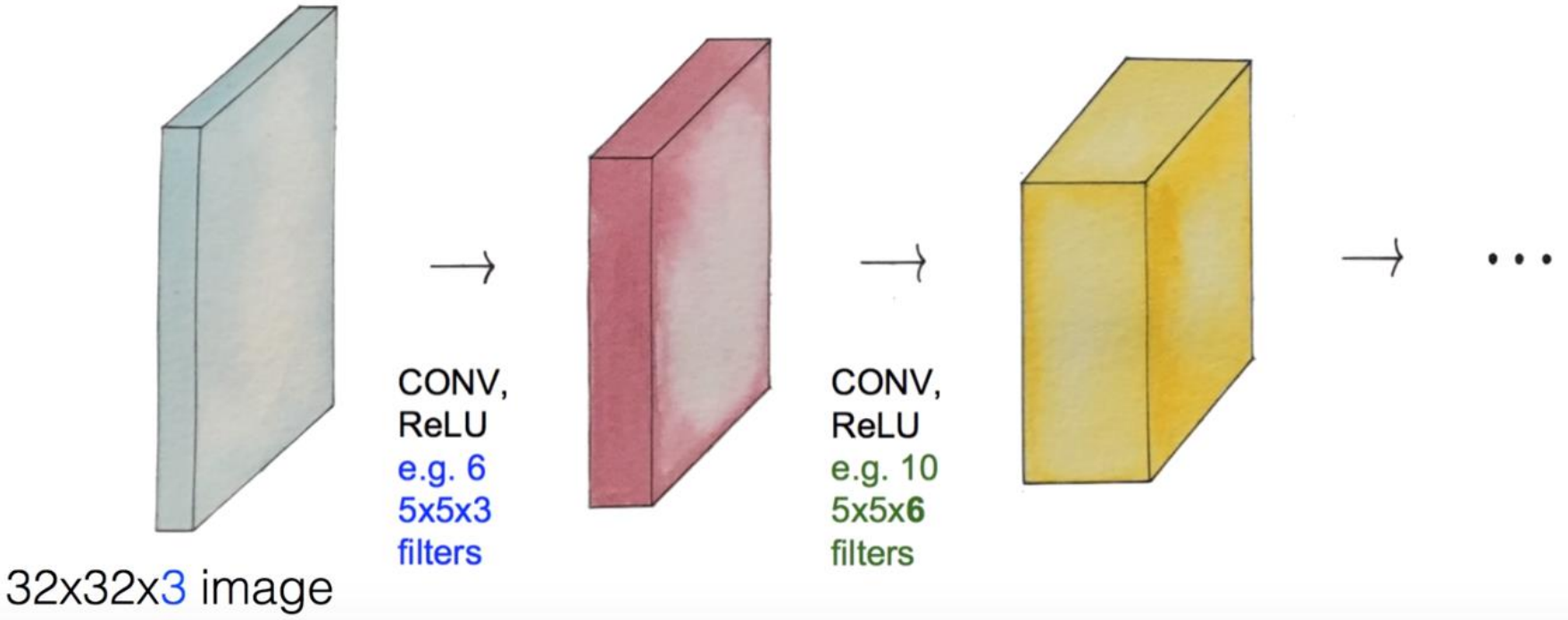
CONV,
ReLU
e.g. 6
5x5x3
filters

Convolution Layers

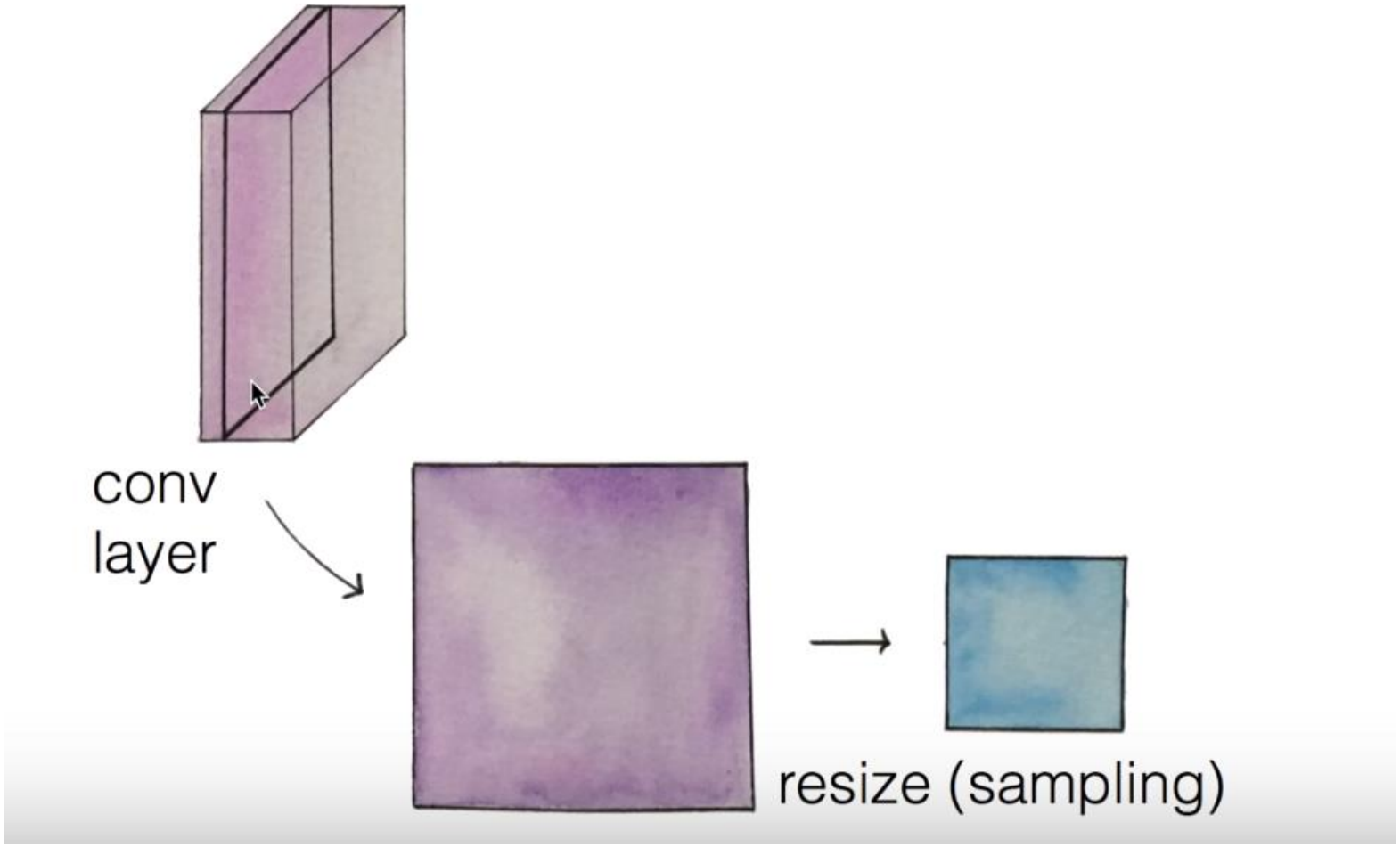


Convolution Layers

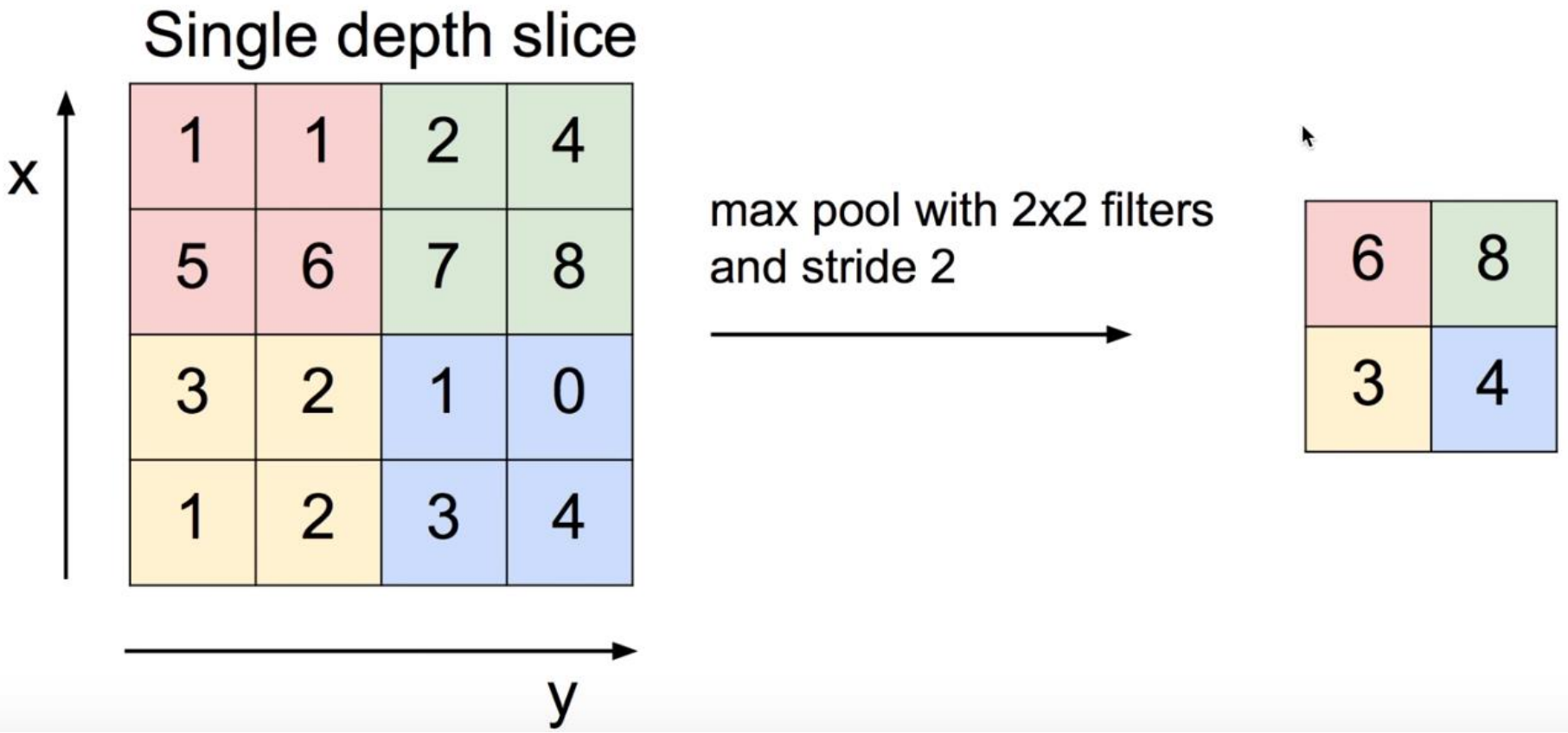
How many weight variables? How to set them?



Pooling Layer (Sampling)

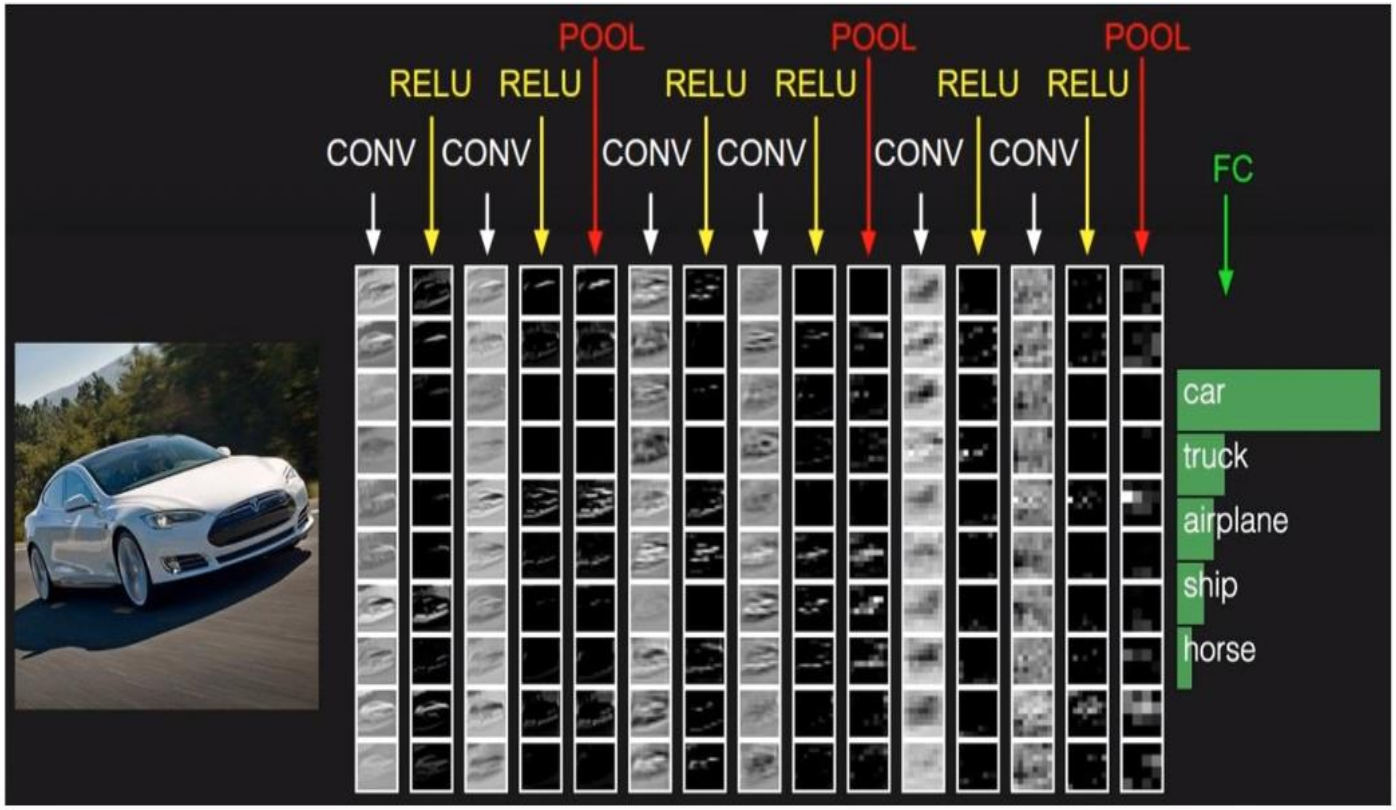


Max Pooling

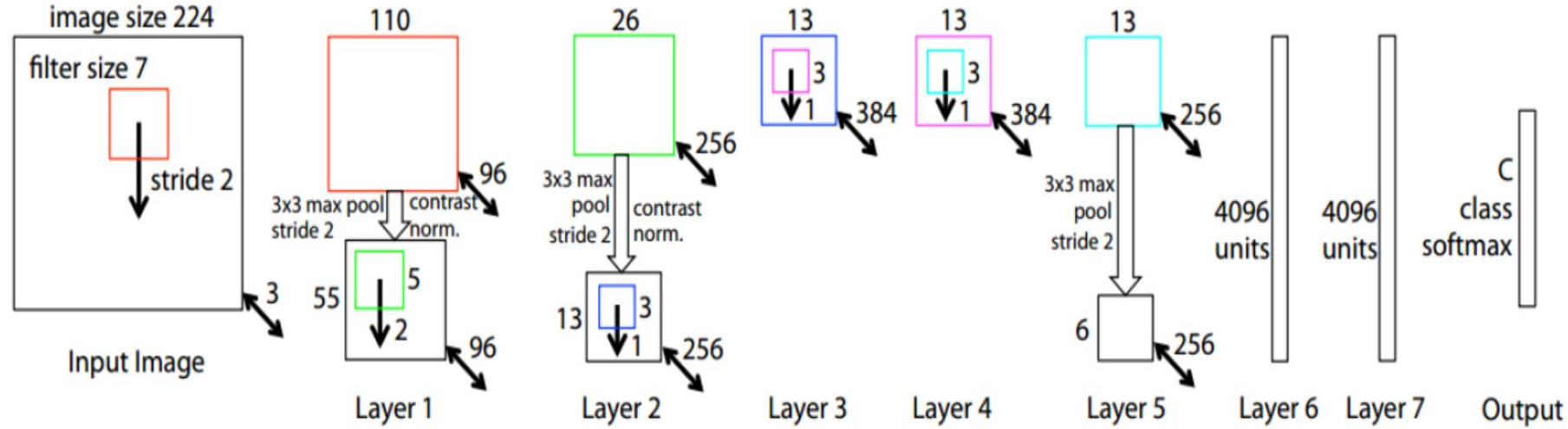

Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

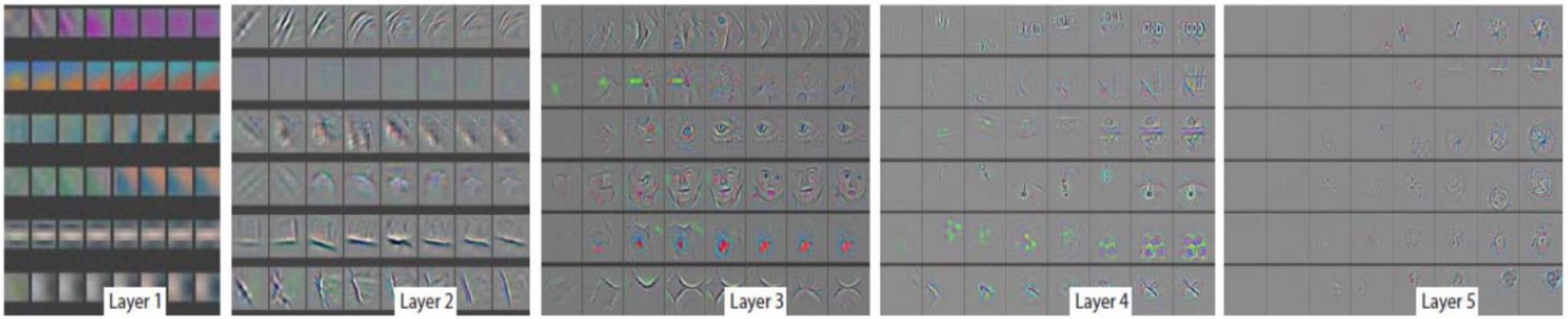
Fully Connected Layer (FC Layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



Deep CNN for Image Classification





CNN by LeCun

- Neural network with specialized connectivity structure
- Stack multiple stages of feature extractors
- Higher stages compute more global, more invariant features
- Classification layer at the end