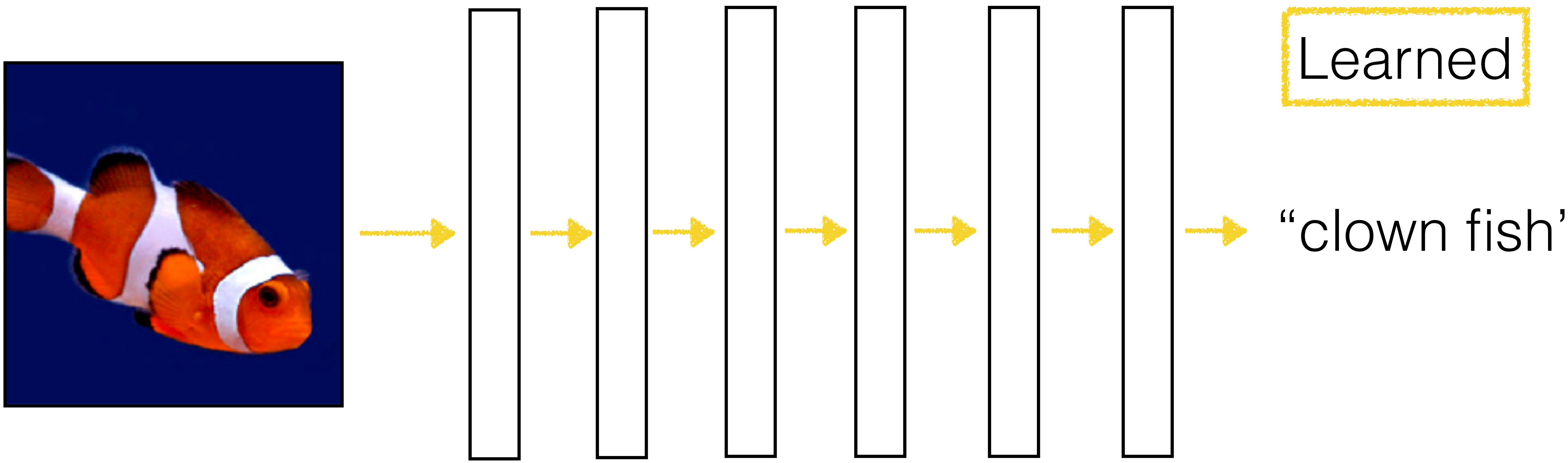
# 95-865 Unstructured Data Analytics

## Week 6: Deep learning for analyzing images and time series, wrap-up

George Chen

# Deep Learning




- Inspired by biological neural nets *but otherwise not the same at all* (biological neural nets do *not* work like deep nets)
- Learns a layered representation
  - Tries to get rid of manual feature engineering
  - Need to design constraints for what features are learned to account for structure in data (e.g., images, text, …)

# Learning a neural net amounts to curve fitting

We’re just estimating a function

# Neural Net as Function Approximation

Given `input`, learn a computer program that computes `output`

this is a **function**

Single-layer neural net example:

```
def f(input):
    output = softmax(np.dot(input, W) + b)
    return output
```

the only things that we are learning
(we fix their dimensions in advance)

We are fixing what the function `f` looks like in code
and are only adjusting `W` and `b`!!!

# Neural Net as Function Approximation

Given `input`, learn a computer program that computes `output`

Single-layer neural net example:

```
output = softmax(np.dot(input, W) + b)
```

Two-layer neural net example:

```
layer1_output = relu(np.dot(input, W1) + b1)

output = softmax(np.dot(layer1_output, W2) + b2)
```

Learning a neural net: learning a simple computer program that maps inputs (raw feature vectors) to outputs (predictions)

# Architecting Neural Nets

- Increasing number of layers (depth) makes neural net more "complex"
  - Learn computer program that has more lines of code
  - Some times, more parameters may be needed
    - If so, more training data may be needed
- Designing neural net architectures is a bit of an art
  - How to select the number of neurons for intermediate layers?
  - Very common in practice: modify existing architectures that are known to work well (e.g., ResNet for computer vision/image processing)

# Keras Has Many Models Already

github.com/keras-team/keras/tree/master/examples

## Keras examples directory

### Vision models examples

mnist_mlp.py Trains a simple deep multi-layer perceptron on the MNIST da

mnist_cnn.py Trains a simple convnet on the MNIST dataset.

cifar10_cnn.py Trains a simple deep CNN on the CIFAR10 small images dat

cifar10_cnn_capsule.py Trains a simple CNN-Capsule Network on the CIFA

cifar10_resnet.py Trains a ResNet on the CIFAR10 small images dataset.

conv_lstm.py Demonstrates the use of a convolutional LSTM network.

image_ocr.py Trains a convolutional stack followed by a recurrent stack an
to perform optical character recognition (OCR).

mnist_acgan.py Implementation of AC-GAN (Auxiliary Classifier GAN) on th

mnist_hierarchical_rnn.py Trains a Hierarchical RNN (HRNN) to classify MN

mnist_siamese.py Trains a Siamese multi-layer perceptron on pairs of digit
dataset.

mnist_swwae.py Trains a Stacked What-Where AutoEncoder built on residu
dataset.

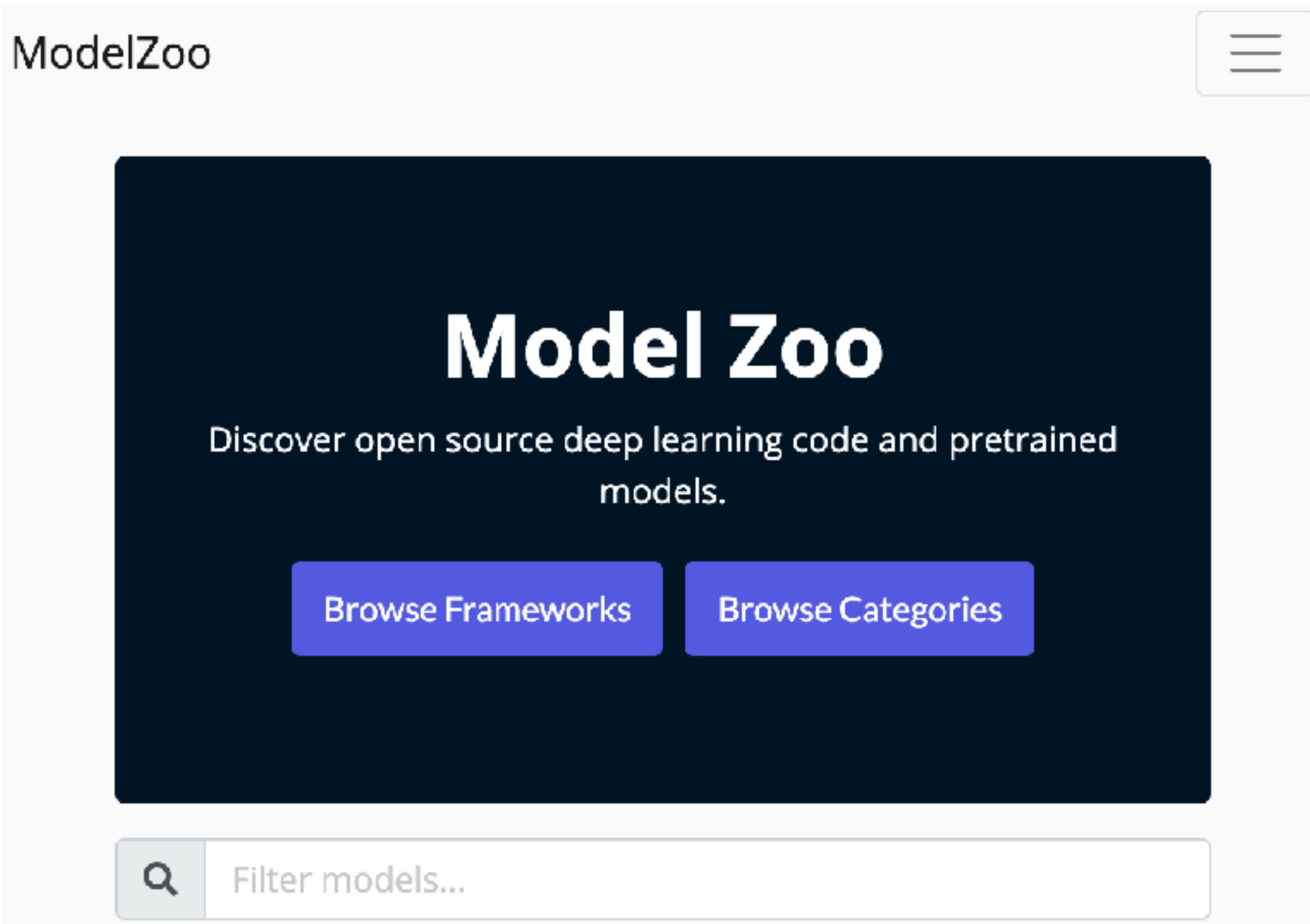
mnist_transfer_cnn.py Transfer learning toy example on the MNIST dataset

mnist_denoising_autoencoder.py Trains a denoising autoencoder on the MI

### Text & sequences examples

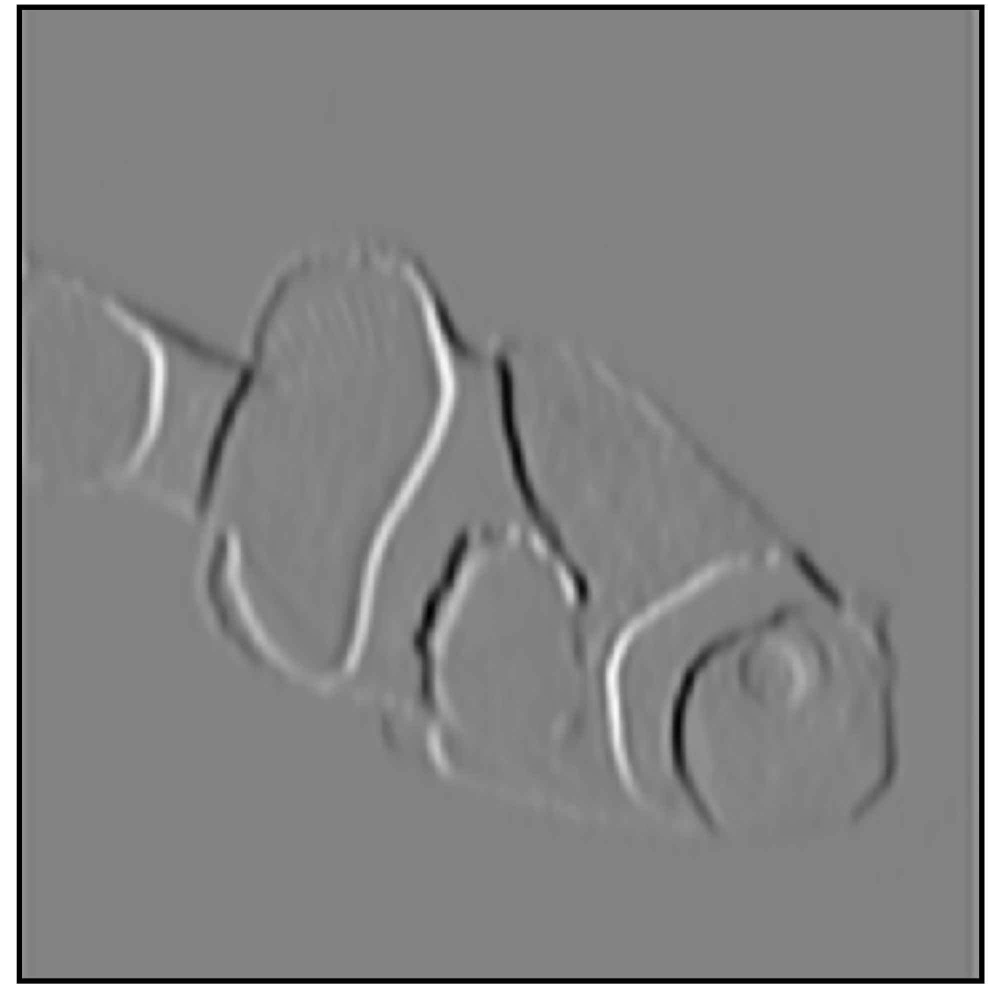
addition_rnn.py Implementation of sequence to sequence learning for perf
numbers (as strings).

# Also check out modelzoo.co

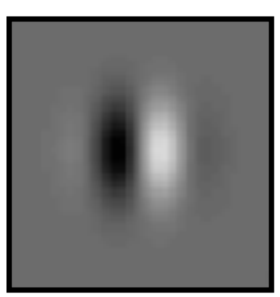


# Image analysis with Convolutional Neural Nets (CNNs, also called convnets)

# Convolution



filter



Slide by Phillip Isola

# Convolution

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

Filter
(also called “kernel”)

# Convolution

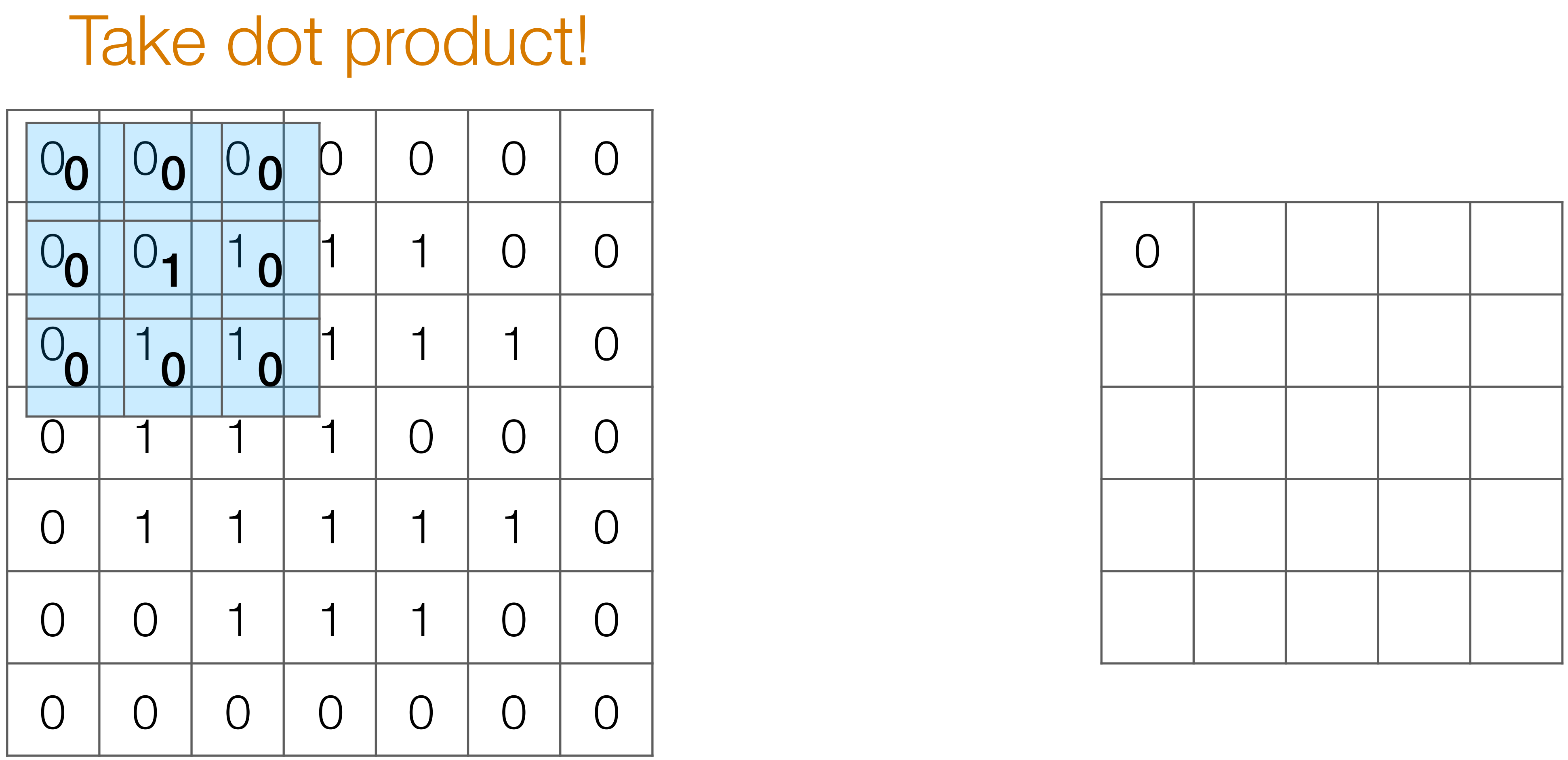
| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

| | | |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 0 | 0 | 0 |

Filter
(also called “kernel”)

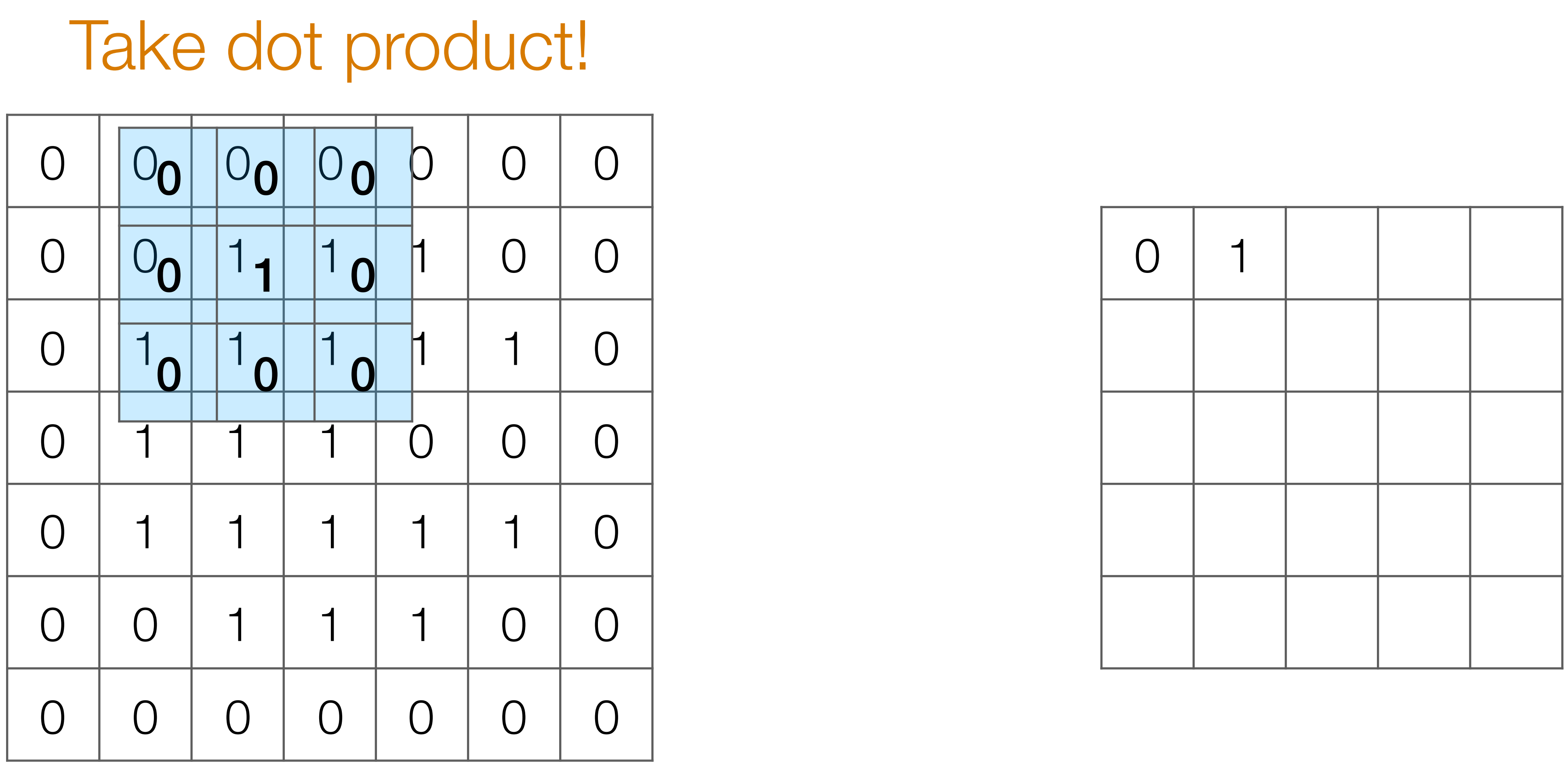
# Convolution




Input image

Output image

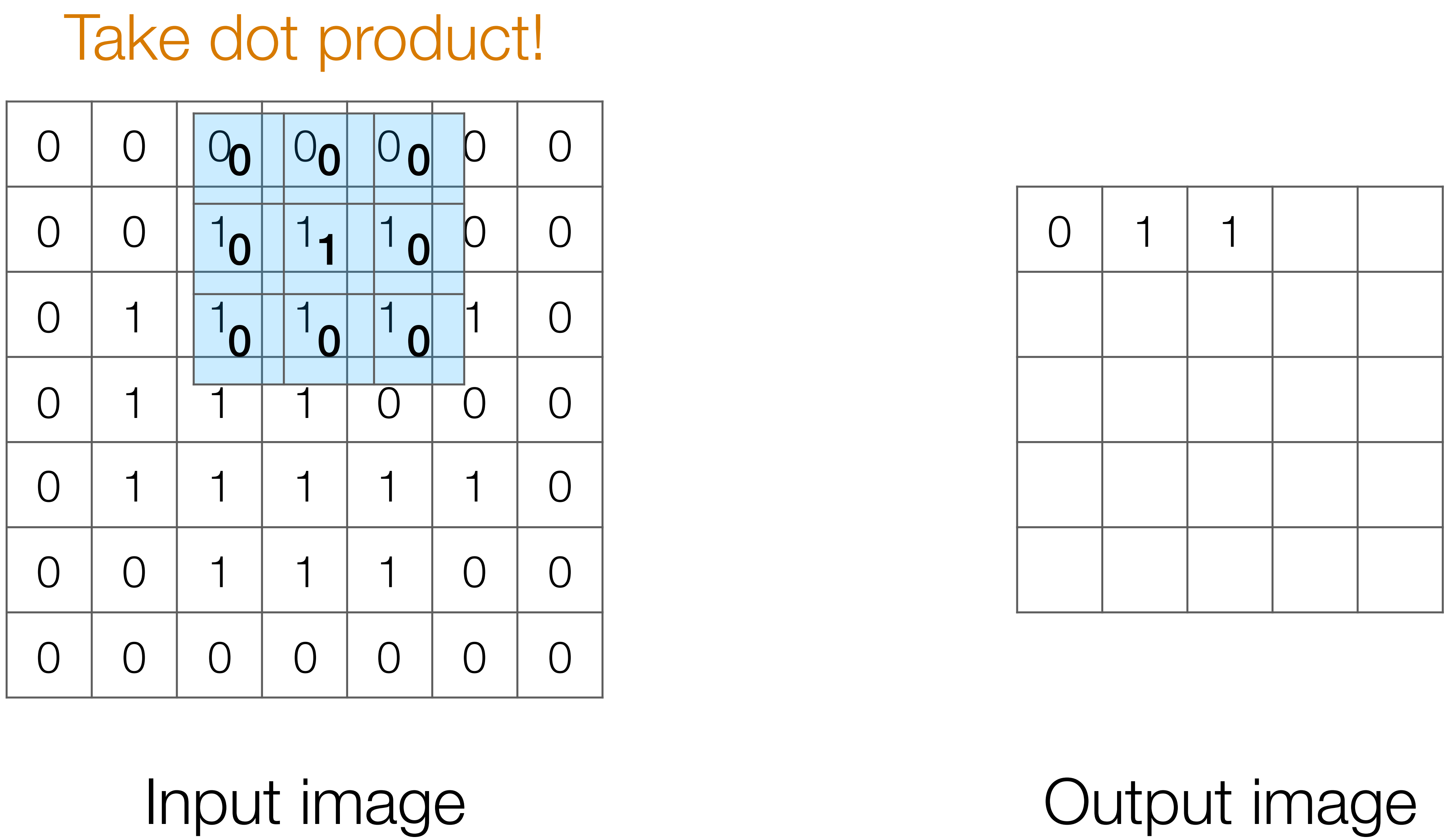
# Convolution




Input image

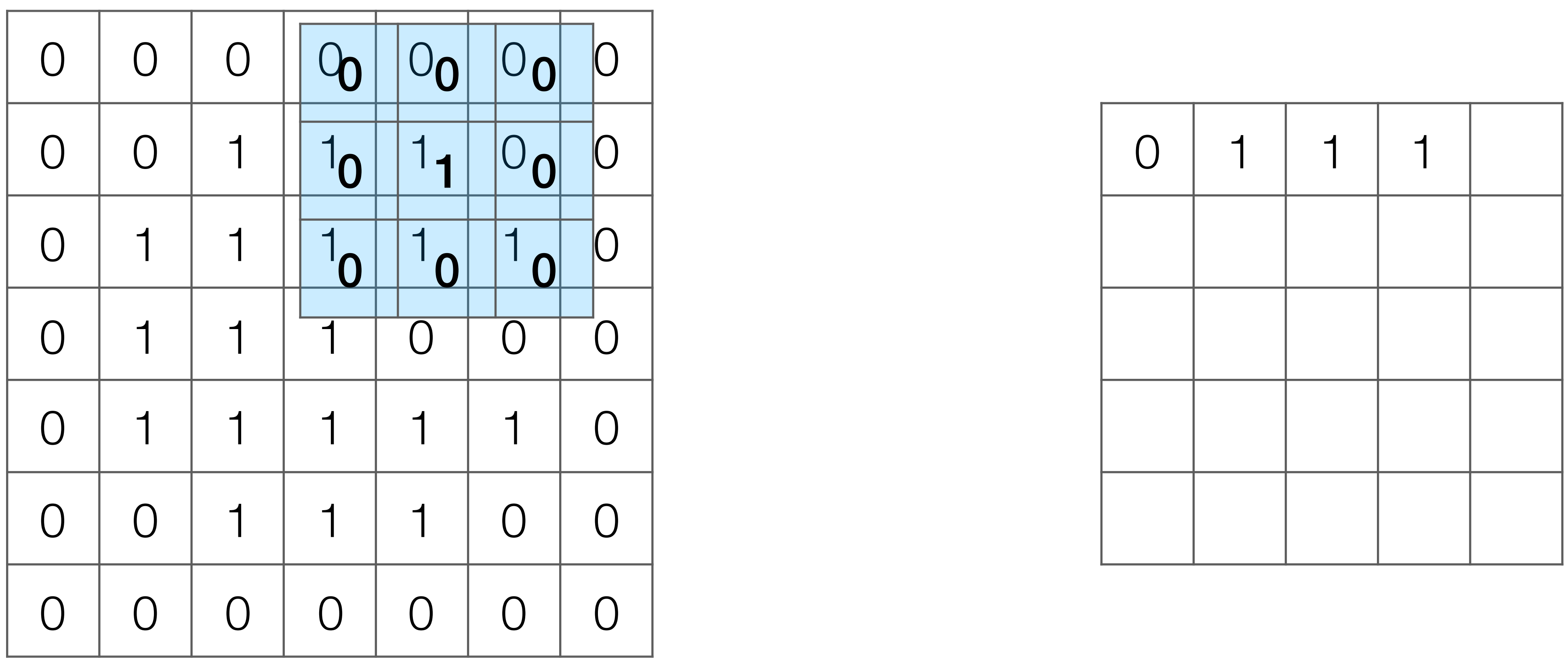
Output image

# Convolution



# Convolution

Take dot product!

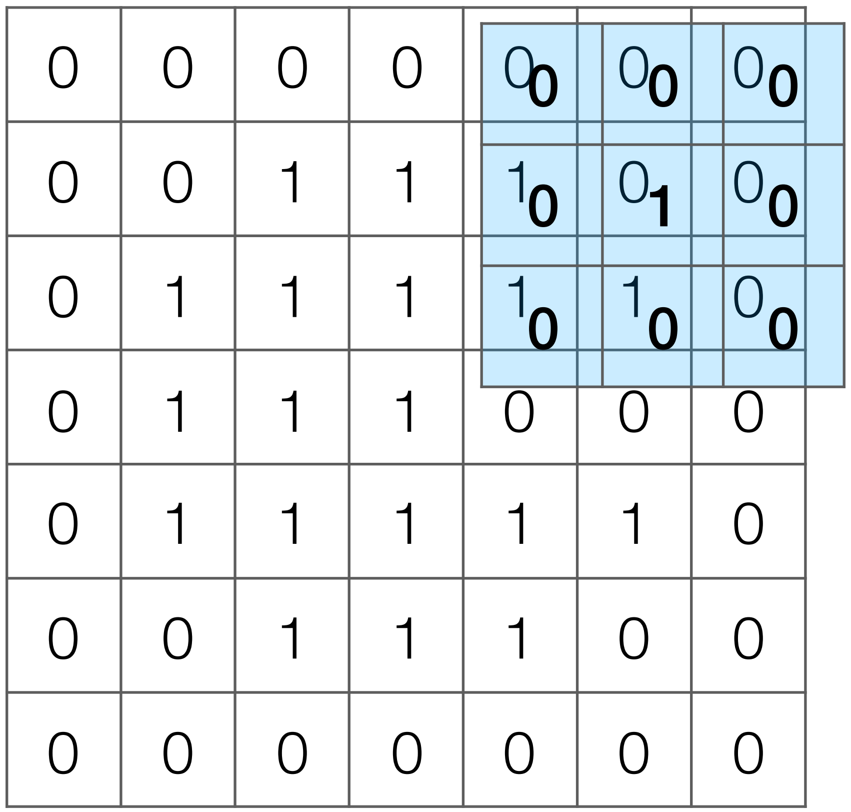



Input image

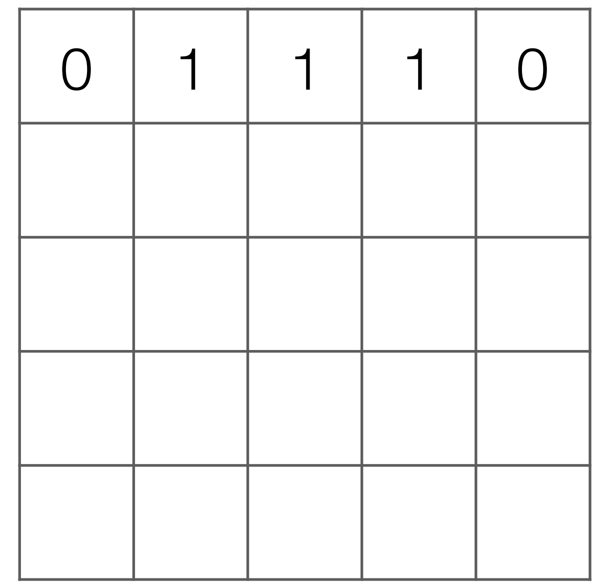
Output image

# Convolution

Take dot product!



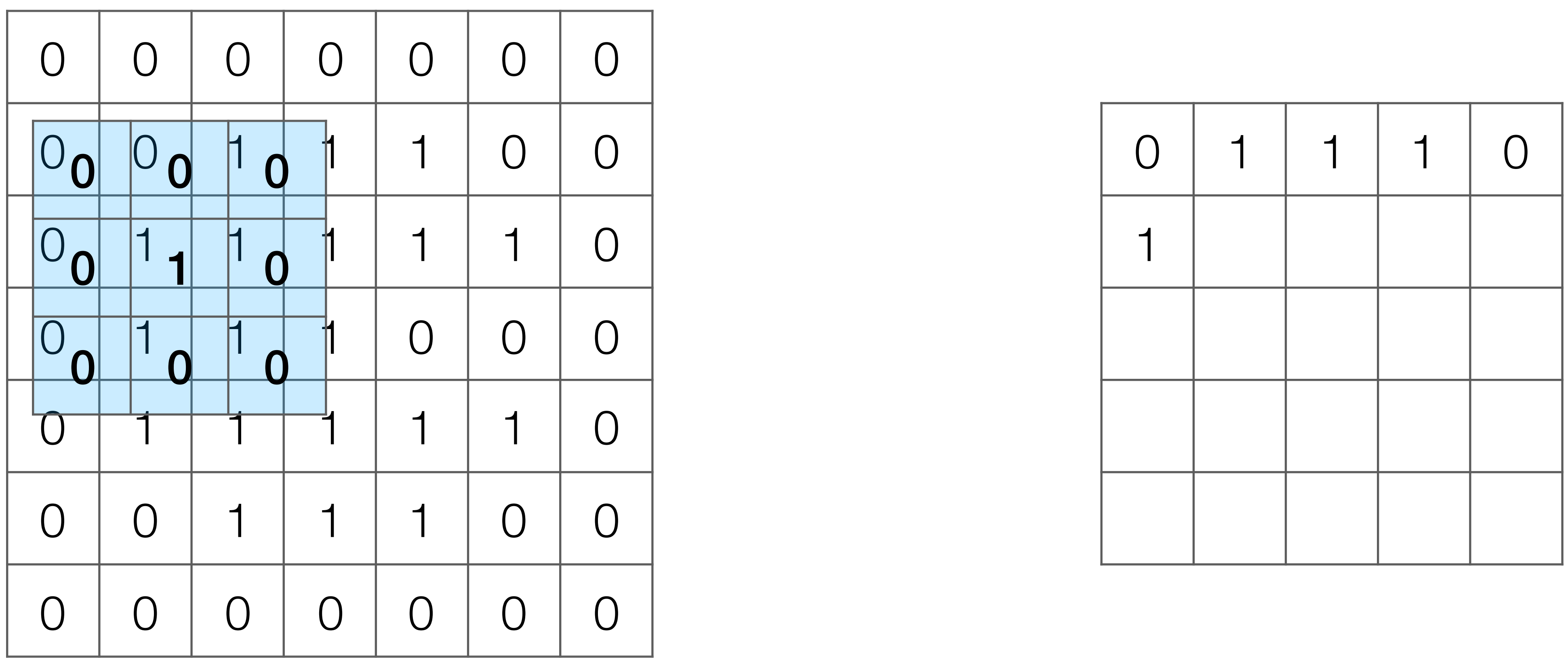

Input image




Output image

# Convolution

Take dot product!

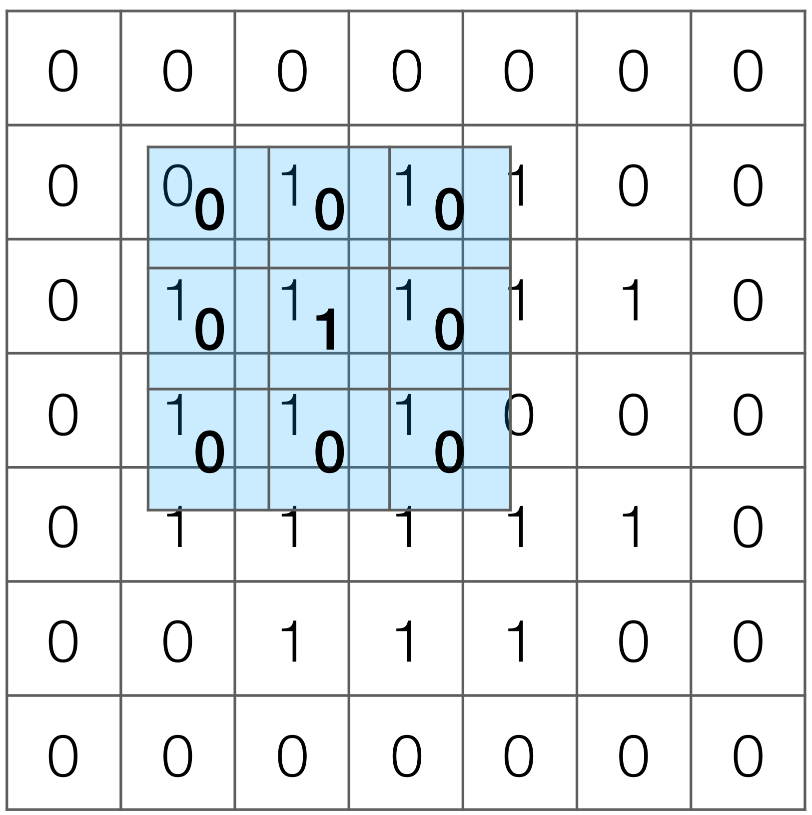


Input image

Output image

# Convolution

Take dot product!



Input image

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | | | |
| | | | | |
| | | | | |
| | | | | |

Output image

# Convolution

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

$*$

| | | |
|---|---|---|
| **0** | **0** | **0** |
| **0** | **1** | **0** |
| **0** | **0** | **0** |

$=$

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |

Output image

Note: output image is smaller than input image

If you want output size to be same as input, pad 0's to input

# Convolution

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

Input image

Output image

Note: output image is smaller than input image

If you want output size to be same as input, pad 0's to input

# Convolution

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

*

| | | |
|---|---|---|
| **0** | **0** | **0** |
| **0** | **1** | **0** |
| **0** | **0** | **0** |

=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 1 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |

Output image

# Convolution

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} * \frac{1}{9} \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} = \frac{1}{9} \begin{bmatrix} 3 & 5 & 6 & 5 & 3 \\ 5 & 8 & 8 & 6 & 3 \\ 6 & 9 & 8 & 7 & 4 \\ 5 & 8 & 8 & 6 & 3 \\ 3 & 5 & 6 & 5 & 3 \end{bmatrix}$$

Input image

Output image

# Convolution

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
*
\begin{bmatrix}
\mathbf{-1} & \mathbf{-1} & \mathbf{-1} \\
\mathbf{2} & \mathbf{2} & \mathbf{2} \\
\mathbf{-1} & \mathbf{-1} & \mathbf{-1}
\end{bmatrix}
=
\begin{bmatrix}
0 & 1 & 3 & 1 & 0 \\
1 & 1 & 1 & 3 & 3 \\
0 & 0 & -2 & -4 & -4 \\
1 & 1 & 1 & 3 & 3 \\
0 & 1 & 3 & 1 & 0
\end{bmatrix}
$$

Input image

Output image

# Convolution

Very commonly used for:

- Blurring an image

*

| | | |
|---|---|---|
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |
| 1/9 | 1/9 | 1/9 |

=

- Finding edges

*

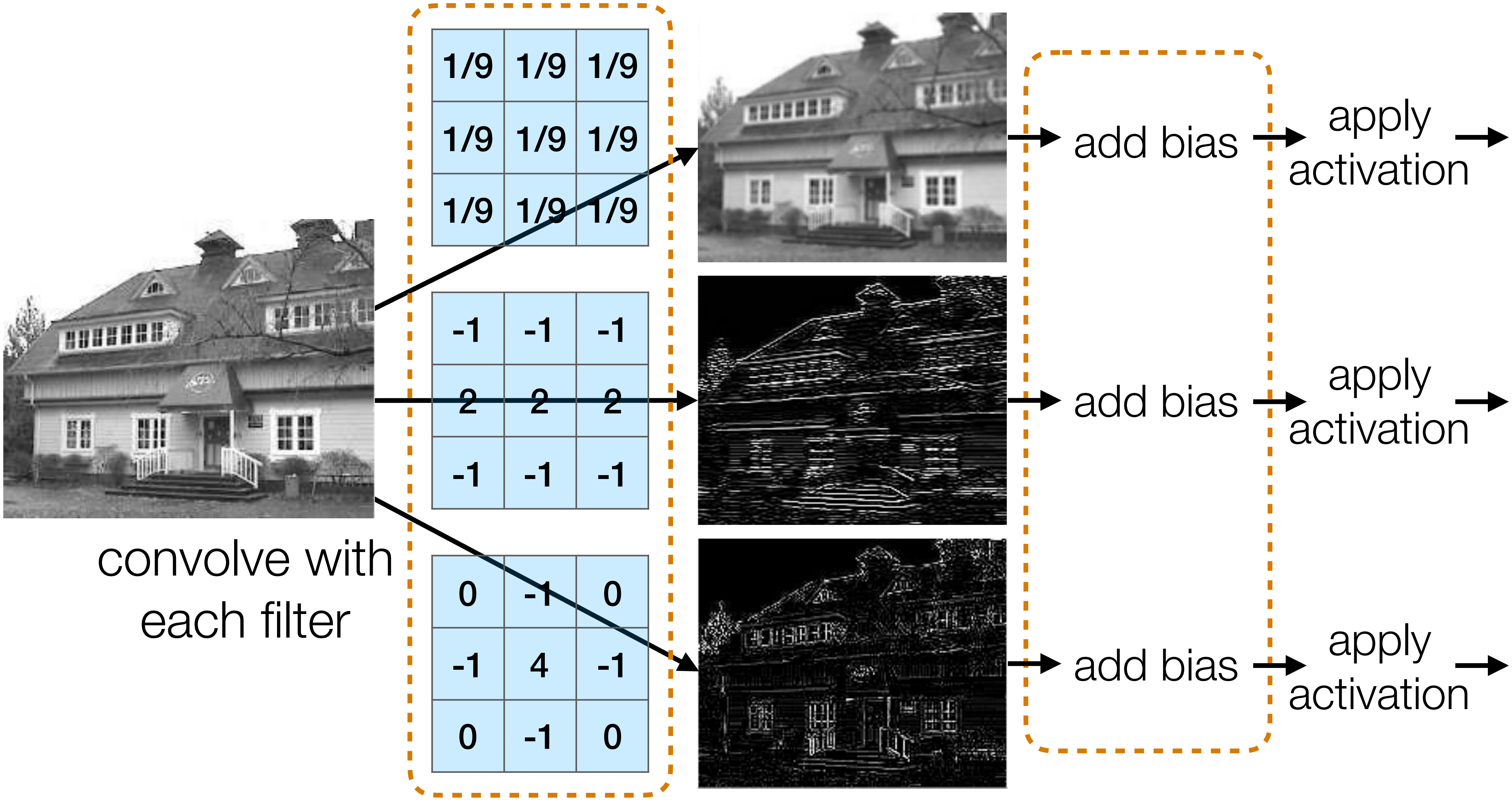
| | | |
|---|---|---|
| -1 | -1 | -1 |
| 2 | 2 | 2 |
| -1 | -1 | -1 |

=

(this example finds horizontal edges)

Images from: http://aishack.in/tutorials/image-convolution-examples/

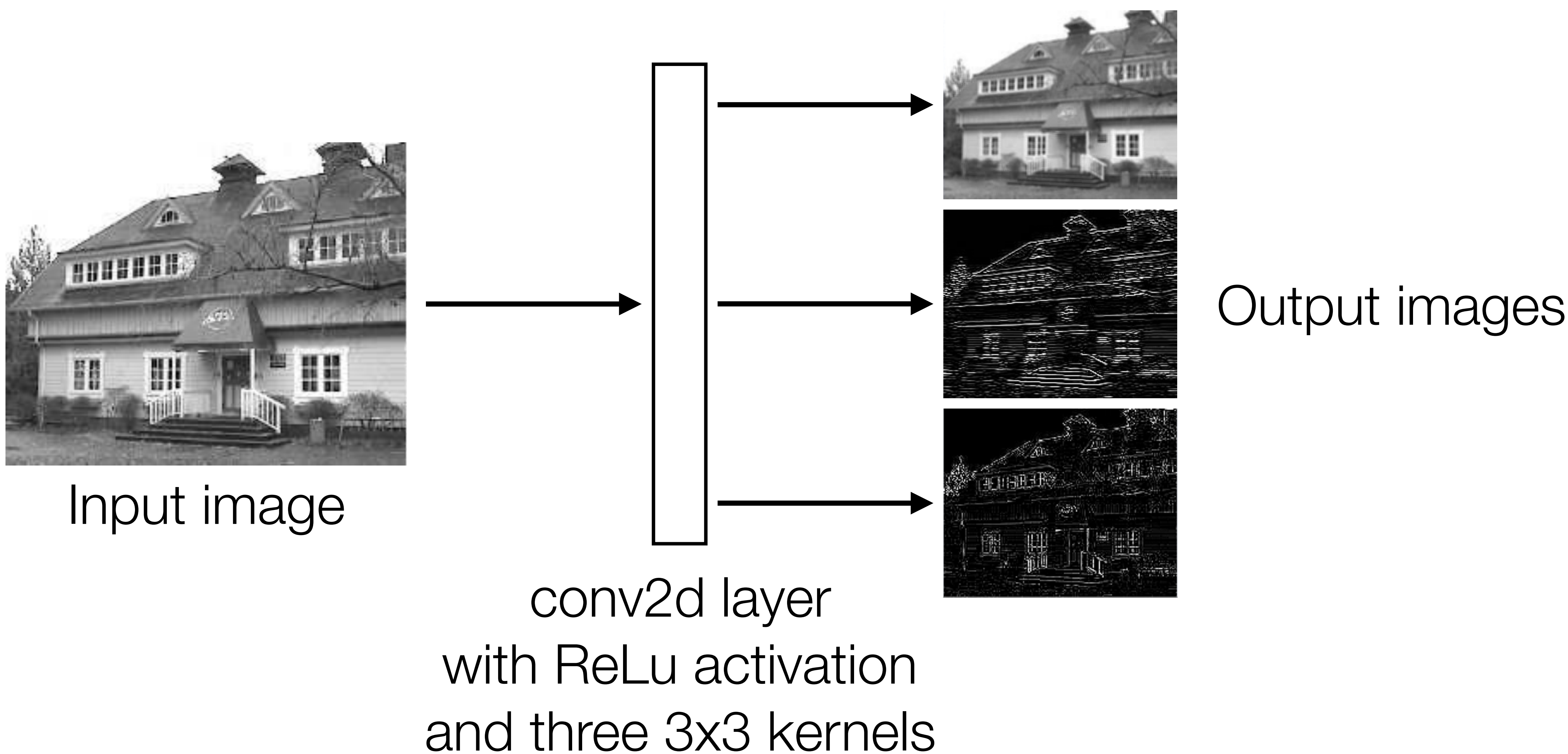
# Convolutional Layer




filters & biases (1 bias number per filter) are unknown and are learned!

Images from: http://aishack.in/tutorials/image-convolution-examples/

# Convolutional Layer




Images from: http://aishack.in/tutorials/image-convolution-examples/

# Convolutional Layer

Input image

dimensions:
height,
width

conv2d layer
with ReLu activation
and three 3x3 kernels

Stack output images into a single “output feature map”

dimensions:
height-2,
width-2,
number of kernels
(3 in this case)

Images from: http://aishack.in/tutorials/image-convolution-examples/

# Convolutional Layer

Input image

dimensions:
height,
width

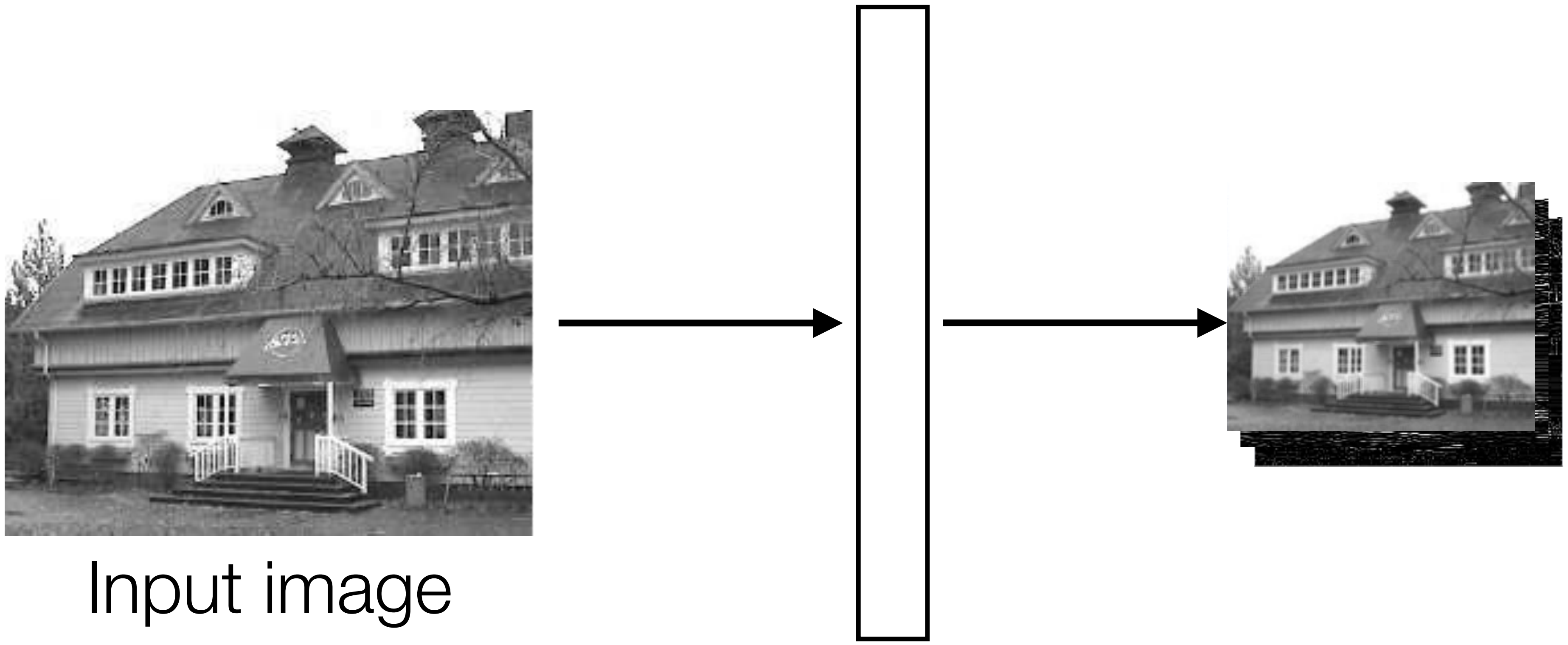
conv2d layer
with ReLu activation
and *k* 3x3 kernels

Stack output images into a single "output feature map"

dimensions:
height-2,
width-2,
*k*

Images from: http://aishack.in/tutorials/image-convolution-examples/

# Convolutional Layer



Stack output images into a single “output feature map”

Input image

dimensions:
height,
width,
depth *d* (# channels)

conv2d layer
with ReLu activation
and *k* 3x3x*d* kernels

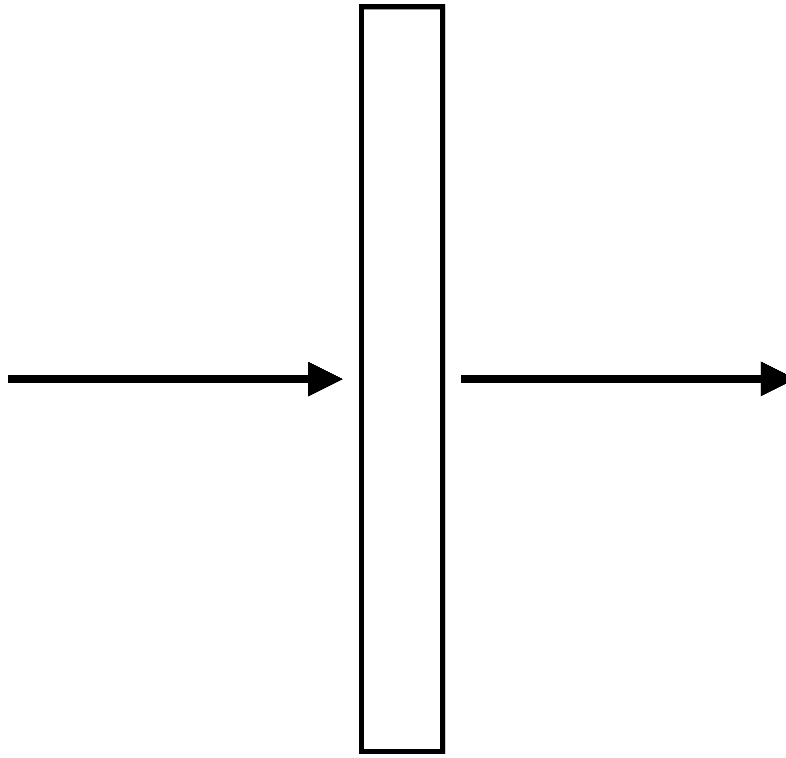
dimensions:
height-2,
width-2,
*k*

Images from: http://aishack.in/tutorials/image-convolution-examples/

# Convolutional Layer

Input image

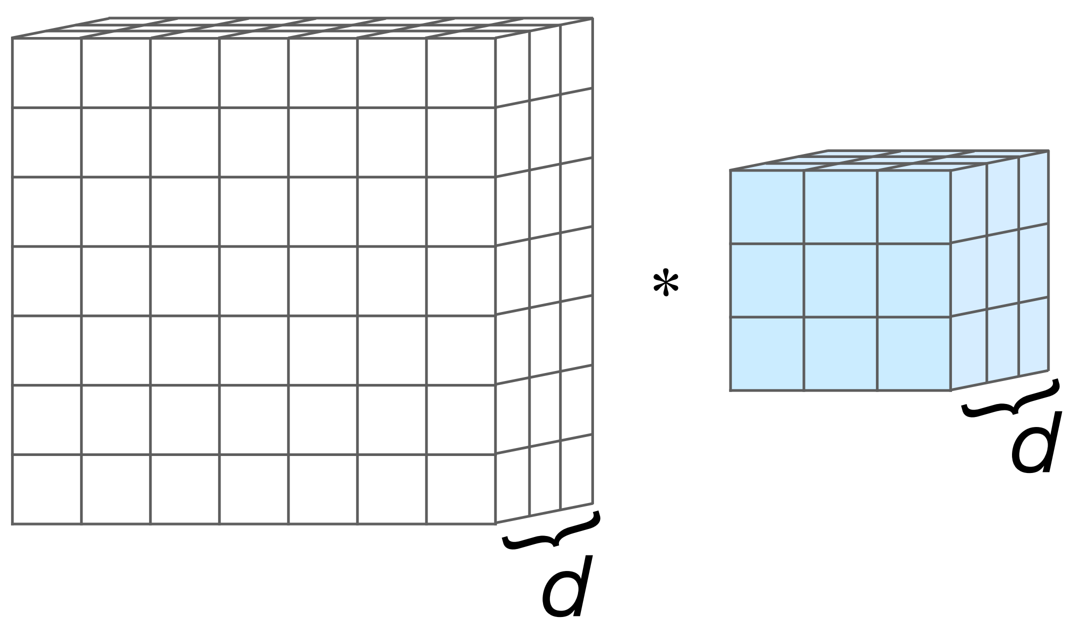
dimensions:
height,
width,
depth *d* (# channels)



conv2d layer
with ReLu activation
and *k* 3x3x*d* kernels

Stack output images into a single "output feature map"

dimensions:
height-2,
width-2,
*k*



# Pooling

- Aggregate local information ("pool" together information)

- Produces a smaller image
  (each resulting pixel captures some "global" information)

- If "object" in input image shifts a little, output is the same

# Max Pooling

Convolutional layer (1 filter, for simplicity no bias)

$$
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
*
\begin{bmatrix}
\mathbf{-1} & \mathbf{-1} & \mathbf{-1} \\
\mathbf{2} & \mathbf{2} & \mathbf{2} \\
\mathbf{-1} & \mathbf{-1} & \mathbf{-1}
\end{bmatrix}
=
\begin{bmatrix}
0 & 1 & 3 & 1 & 0 \\
1 & 1 & 1 & 3 & 3 \\
0 & 0 & -2 & -4 & -4 \\
1 & 1 & 1 & 3 & 3 \\
0 & 1 & 3 & 1 & 0
\end{bmatrix}
$$

Input image

# Max Pooling

Convolutional layer (1 filter, for simplicity no bias)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

*

| | | |
|---|---|---|
| **-1** | **-1** | **-1** |
| **2** | **2** | **2** |
| **-1** | **-1** | **-1** |

=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | -2 | -4 | -4 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

Output image after ReLU

| | |
|---|---|
| | |
| | |

Output after max pooling

# Max Pooling

Convolutional layer (1 filter, for simplicity no bias)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

*

| | | |
|---|---|---|
| **-1** | **-1** | **-1** |
| **2** | **2** | **2** |
| **-1** | **-1** | **-1** |

=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | -2 | -4 | -4 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

Output image after ReLU

| | |
|---|---|
| 1 | |
| | |

Output after max pooling

# Max Pooling

Convolutional layer (1 filter, for simplicity no bias)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

*

| | | |
|---|---|---|
| **-1** | **-1** | **-1** |
| **2** | **2** | **2** |
| **-1** | **-1** | **-1** |

=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | -2 | -4 | -4 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

Output image after ReLU

| | |
|---|---|
| 1 | 3 |
| | |

Output after max pooling

# Max Pooling

Convolutional layer (1 filter, for simplicity no bias)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

*

| | | |
|---|---|---|
| **-1** | **-1** | **-1** |
| **2** | **2** | **2** |
| **-1** | **-1** | **-1** |

=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | -2 | -4 | -4 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

Output image after ReLU

| | |
|---|---|
| 1 | 3 |
| 1 | |

Output after max pooling

# Max Pooling

Convolutional layer (1 filter, for simplicity no bias)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

*

| | | |
|---|---|---|
| **-1** | **-1** | **-1** |
| **2** | **2** | **2** |
| **-1** | **-1** | **-1** |

=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | -2 | -4 | -4 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

Output image after ReLU

| | |
|---|---|
| 1 | 3 |
| 1 | 3 |

Output after max pooling

# Max Pooling

Convolutional layer (1 filter, for simplicity no bias)

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Input image

*

| | | |
|---|---|---|
| **-1** | **-1** | **-1** |
| **2** | **2** | **2** |
| **-1** | **-1** | **-1** |

=

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | -2 | -4 | -4 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

| | | | | |
|---|---|---|---|---|
| 0 | 1 | 3 | 1 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 3 | 3 |
| 0 | 1 | 3 | 1 | 0 |

Output image after ReLU

What numbers were involved in computing this 1?

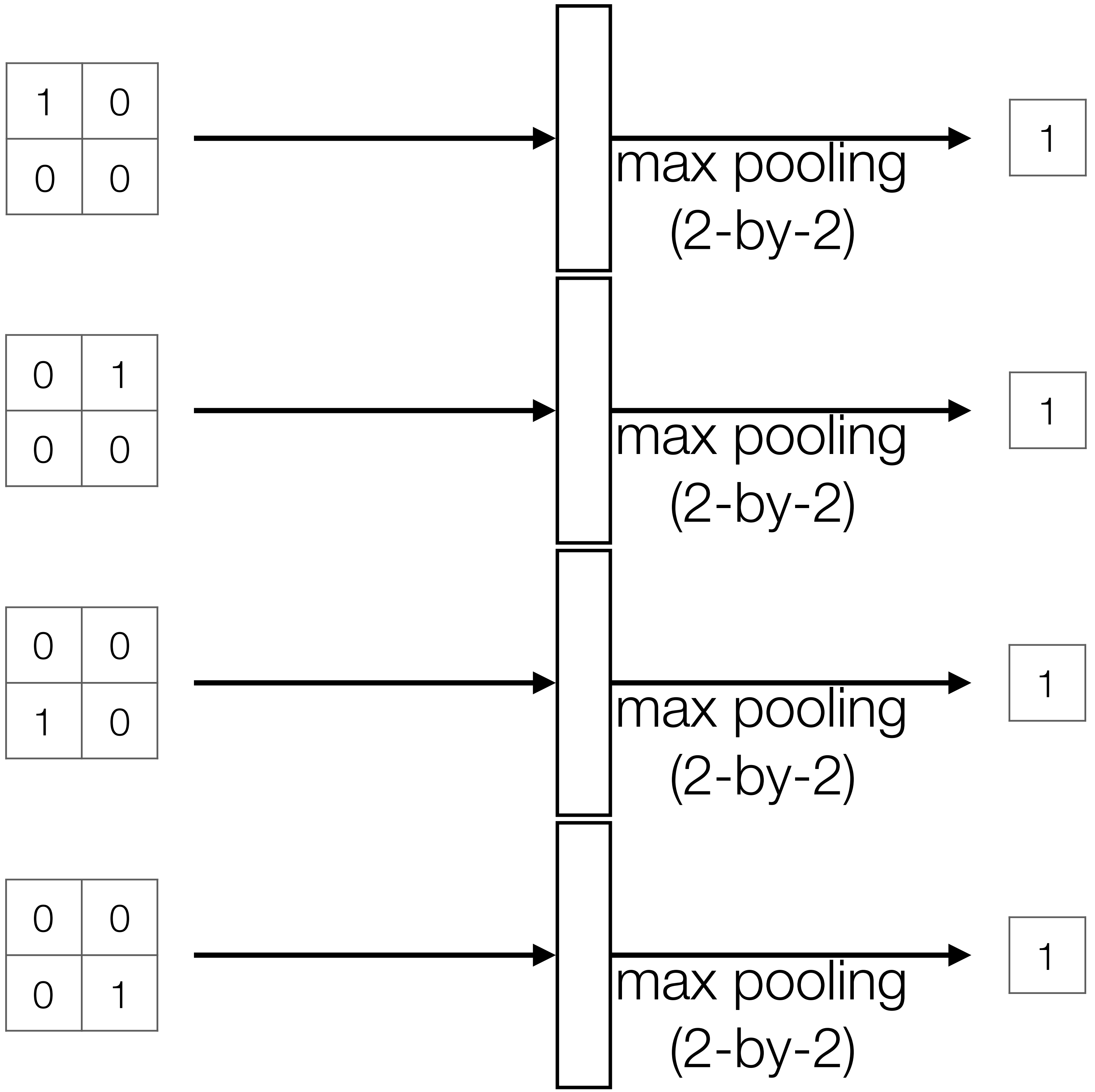
| | |
|---|---|
| 1 | 3 |
| 1 | 3 |

Output after max pooling

In this example: 1 pixel in max pooling output captures information from 16 input pixels!

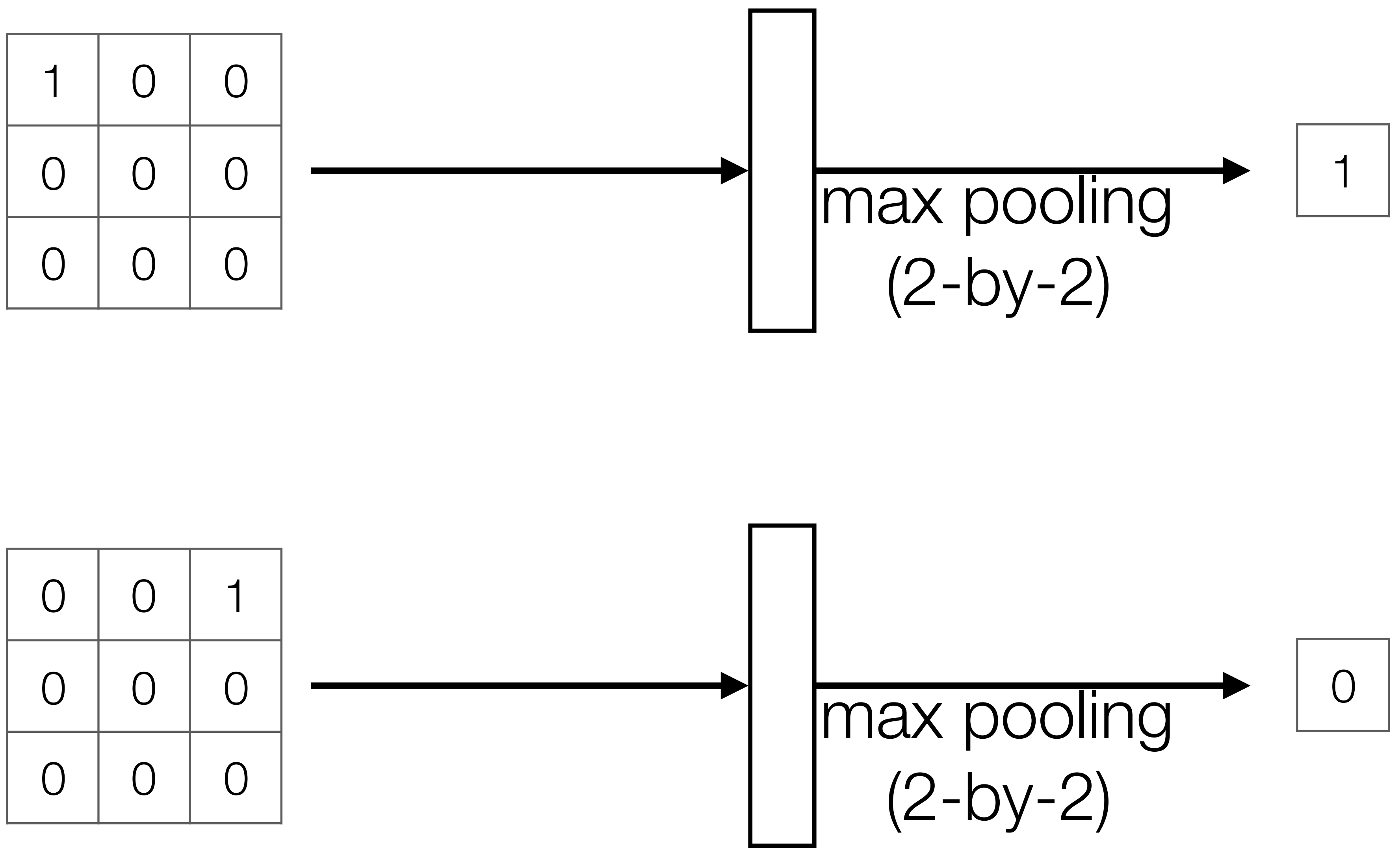
Example: applying max pooling again results in a single pixel that captures info from entire input image!

# Max Pooling and (Slight) Shift Invariance



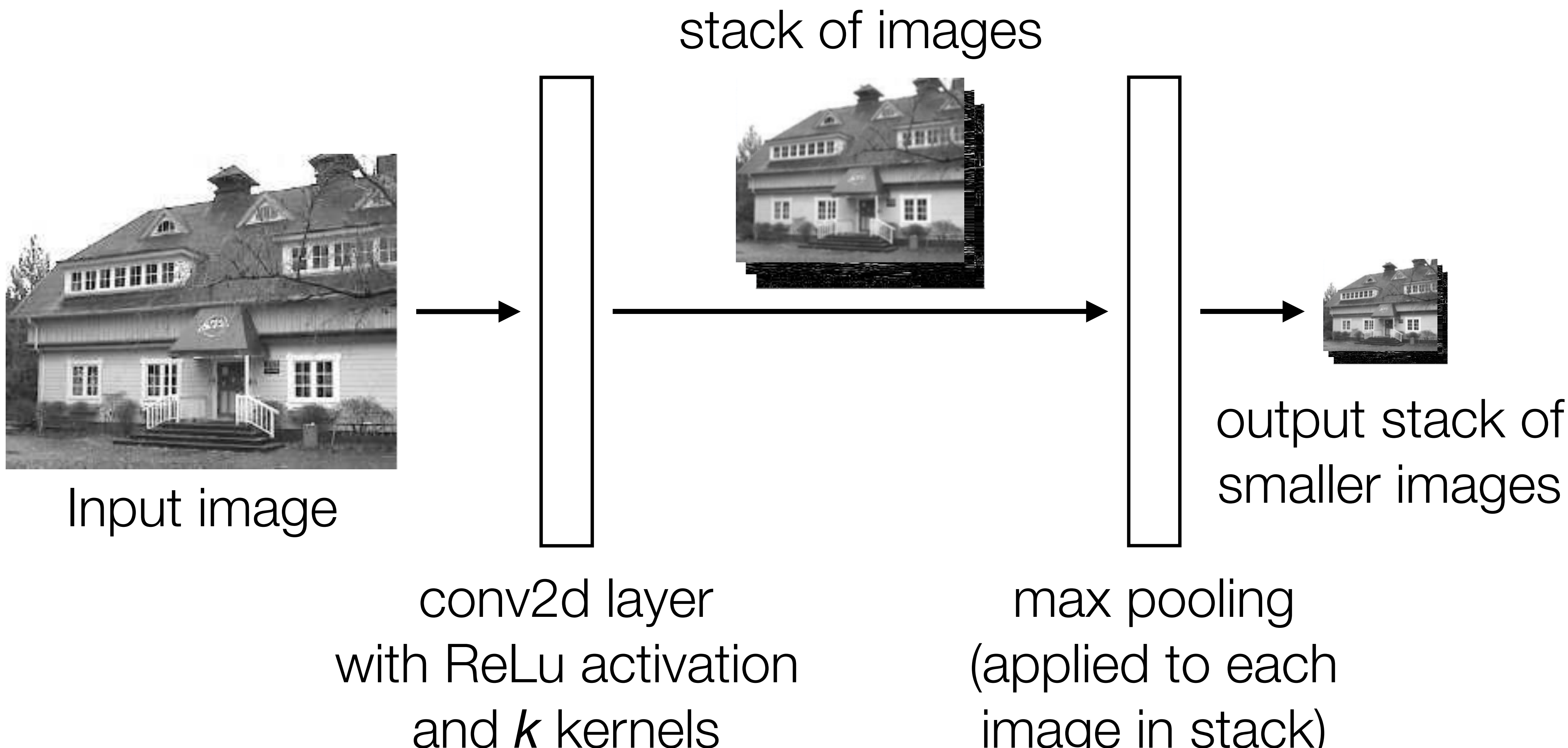

Small shift of "object" (e.g., a detected edge) in input image results in same output

# Max Pooling and (Slight) Shift Invariance



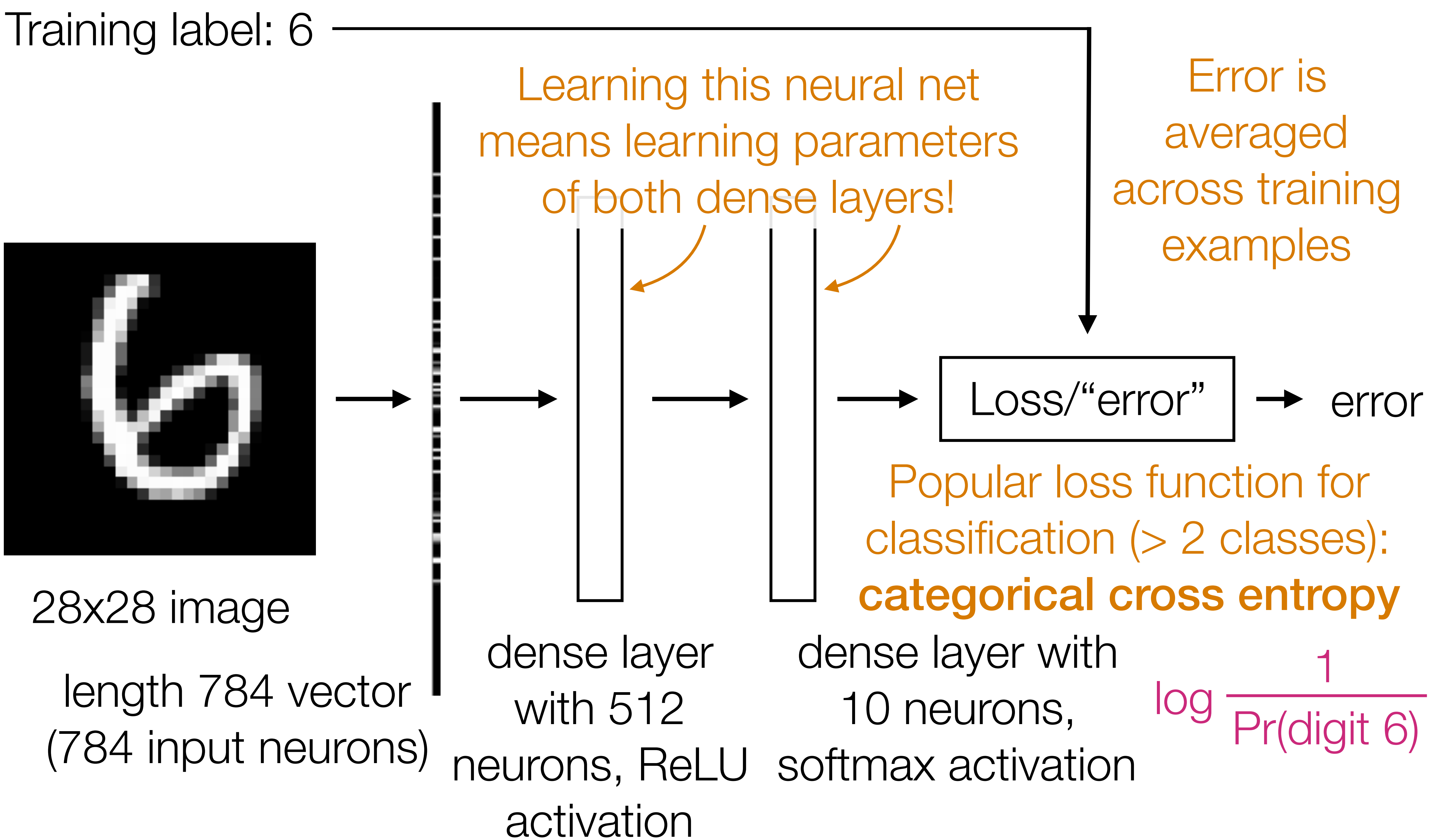

Bigger shift in input can still change output

# Basic Building Block of CNNs

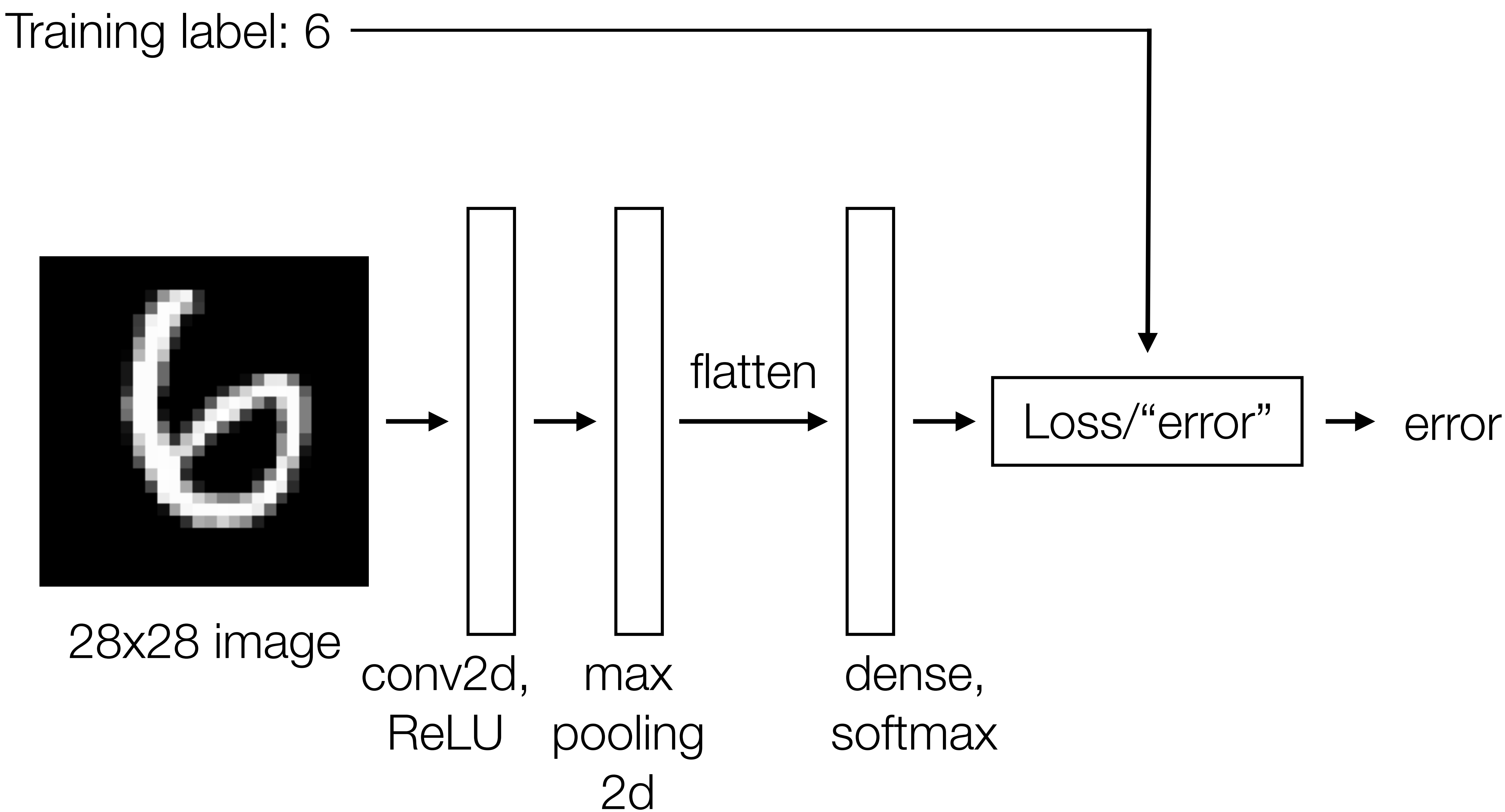



Images from: http://aishack.in/tutorials/image-convolution-examples/

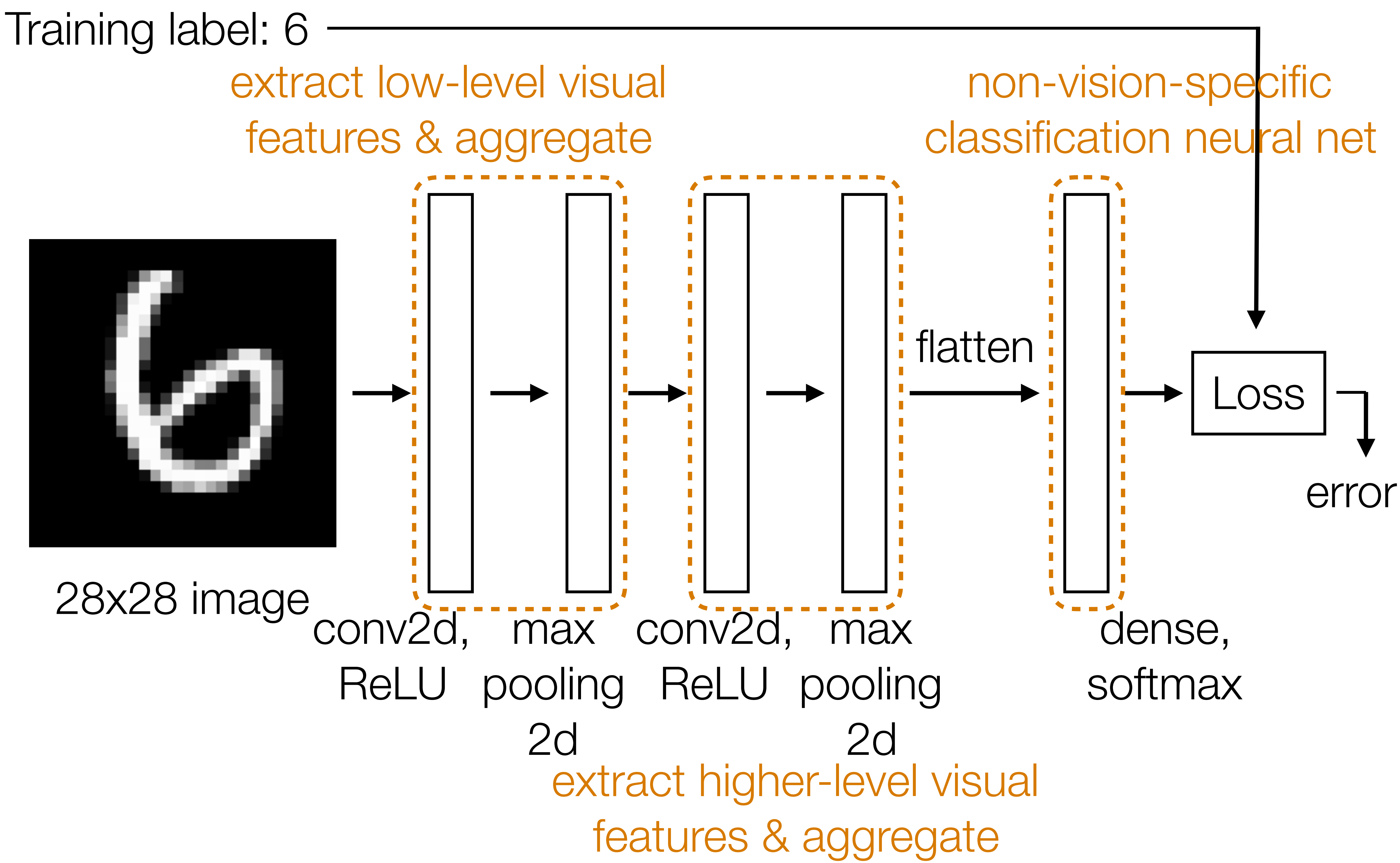
# Handwritten Digit Recognition



# Handwritten Digit Recognition



# Handwritten Digit Recognition



# CNNs

Demo

# CNNs

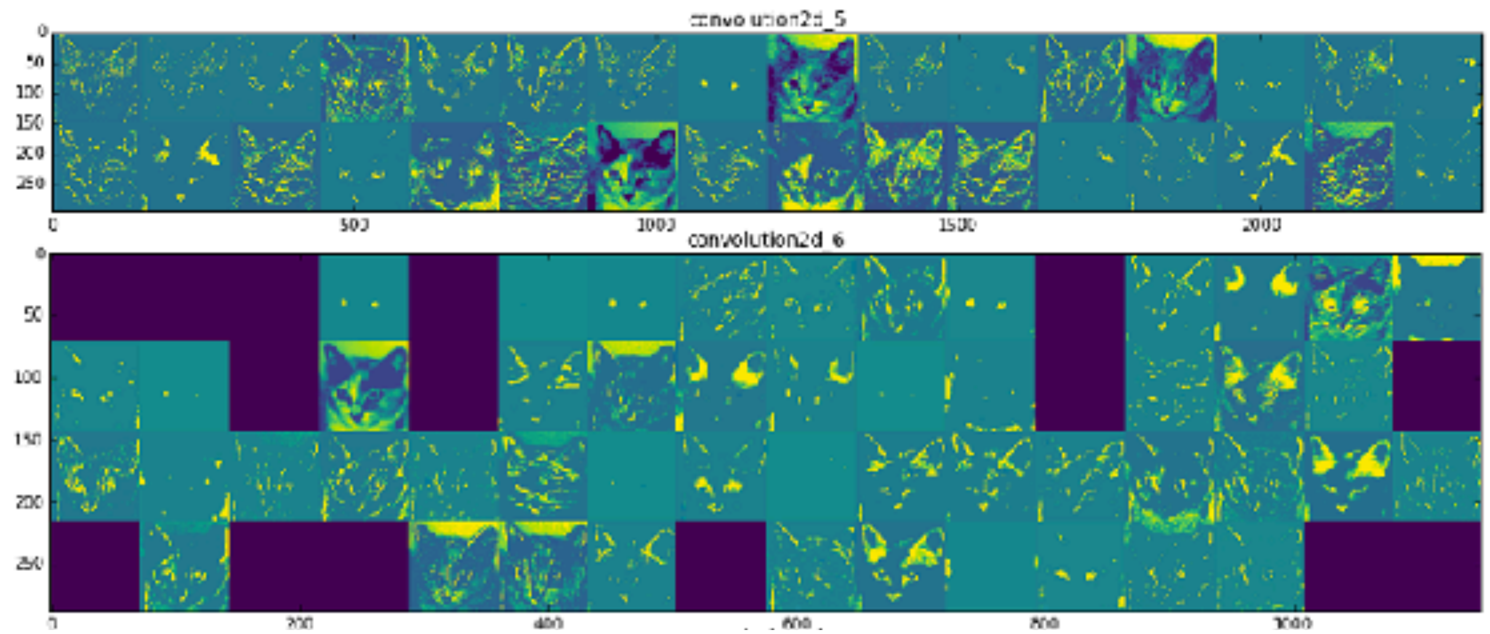
- Learn convolution filters for extracting simple features

- Max pooling summarizes information and produces a *smaller* output and is invariant to small shifts in input objects

- Can then repeat the above two layers to learn features from increasingly higher-level representations

# Visualizing What a Deep Net Learned

- Very straight-forward for CNNs
  - Plot filter outputs at different layers




  - Plot regions that maximally activate an output neuron

Check course webpage for demo

Images: Francois Chollet's "Deep Learning with Python" Chapter 5

# Example: Wolves vs Huskies

(a) Husky classified as wolf (b) Explanation

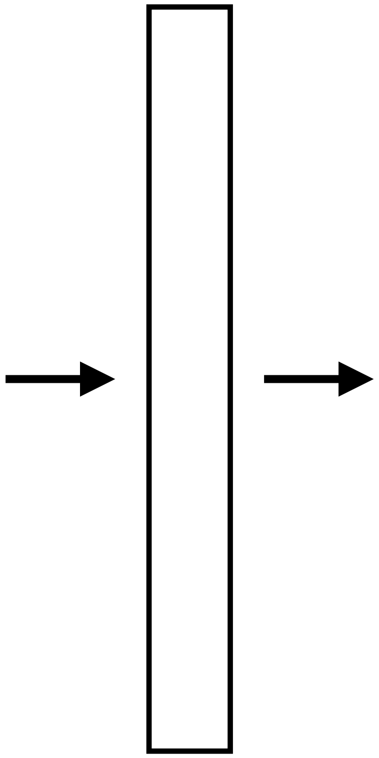
Turns out the deep net learned that wolves are wolves because of snow…

➔ visualization is crucial!

Source: Ribeiro et al. "Why should I trust you? Explaining the predictions of any classifier." KDD 2016.

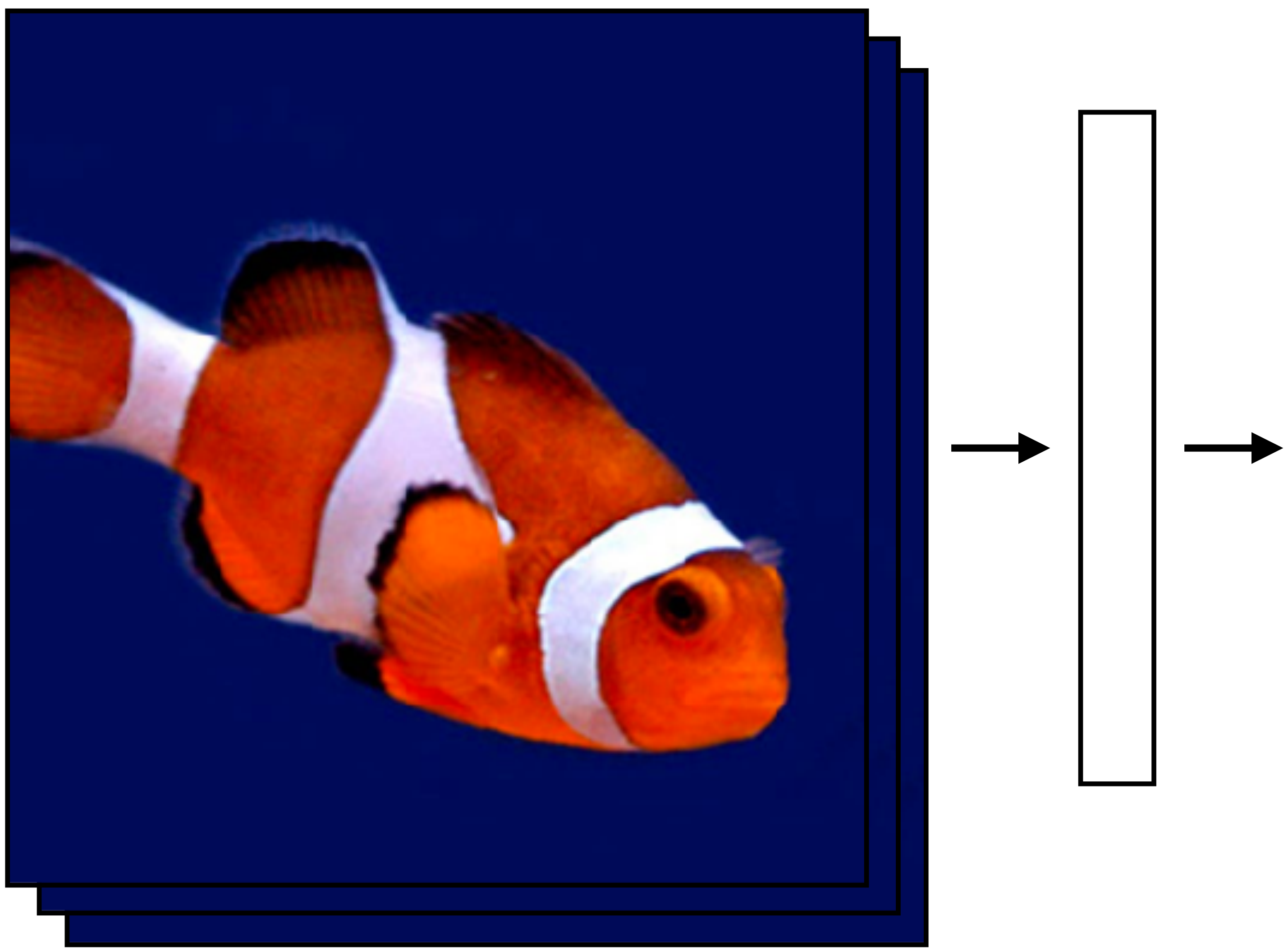
# RNNs

What we've seen so far are "feedforward" NNs



# RNNs

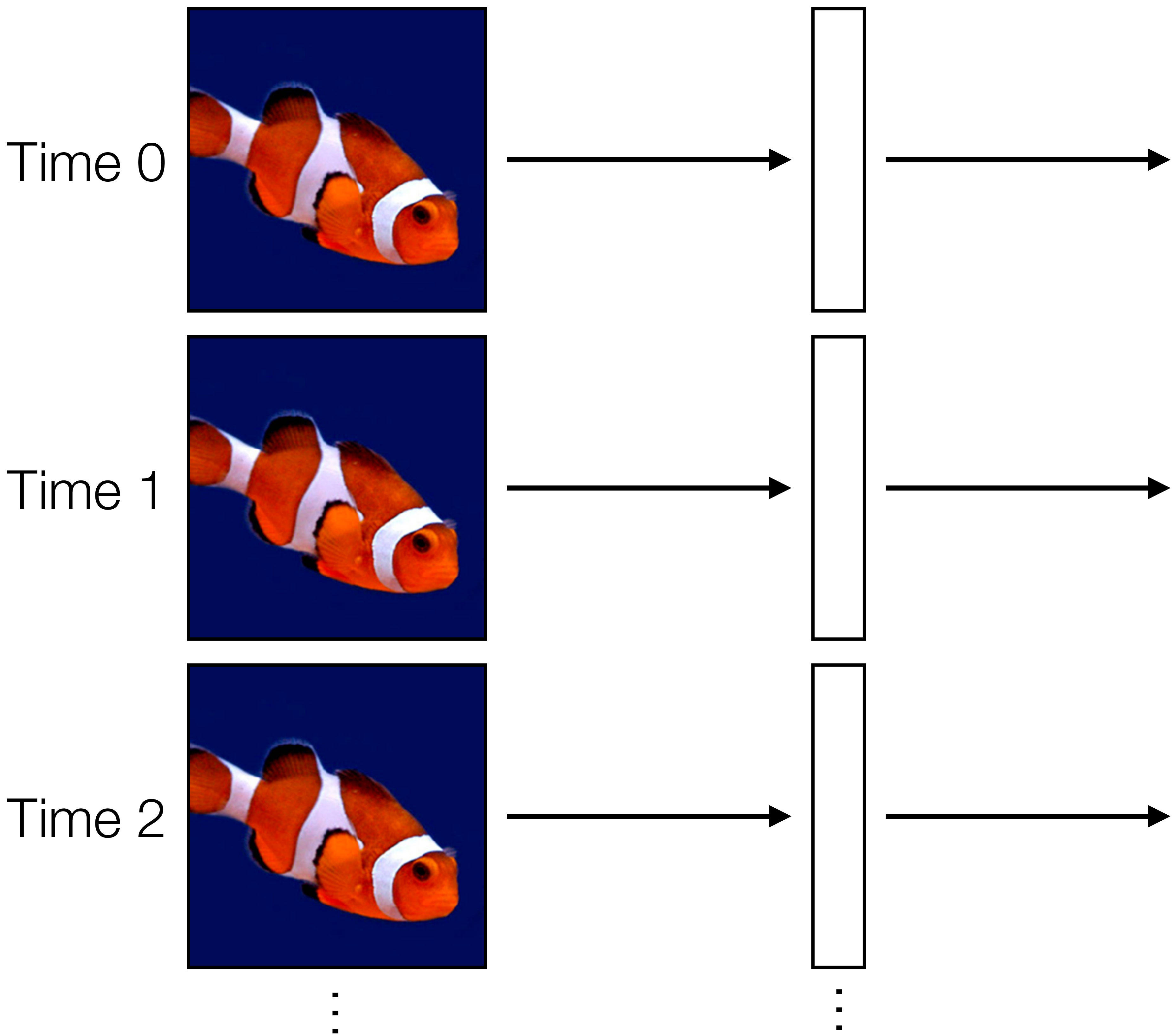
What we've seen so far are "feedforward" NNs



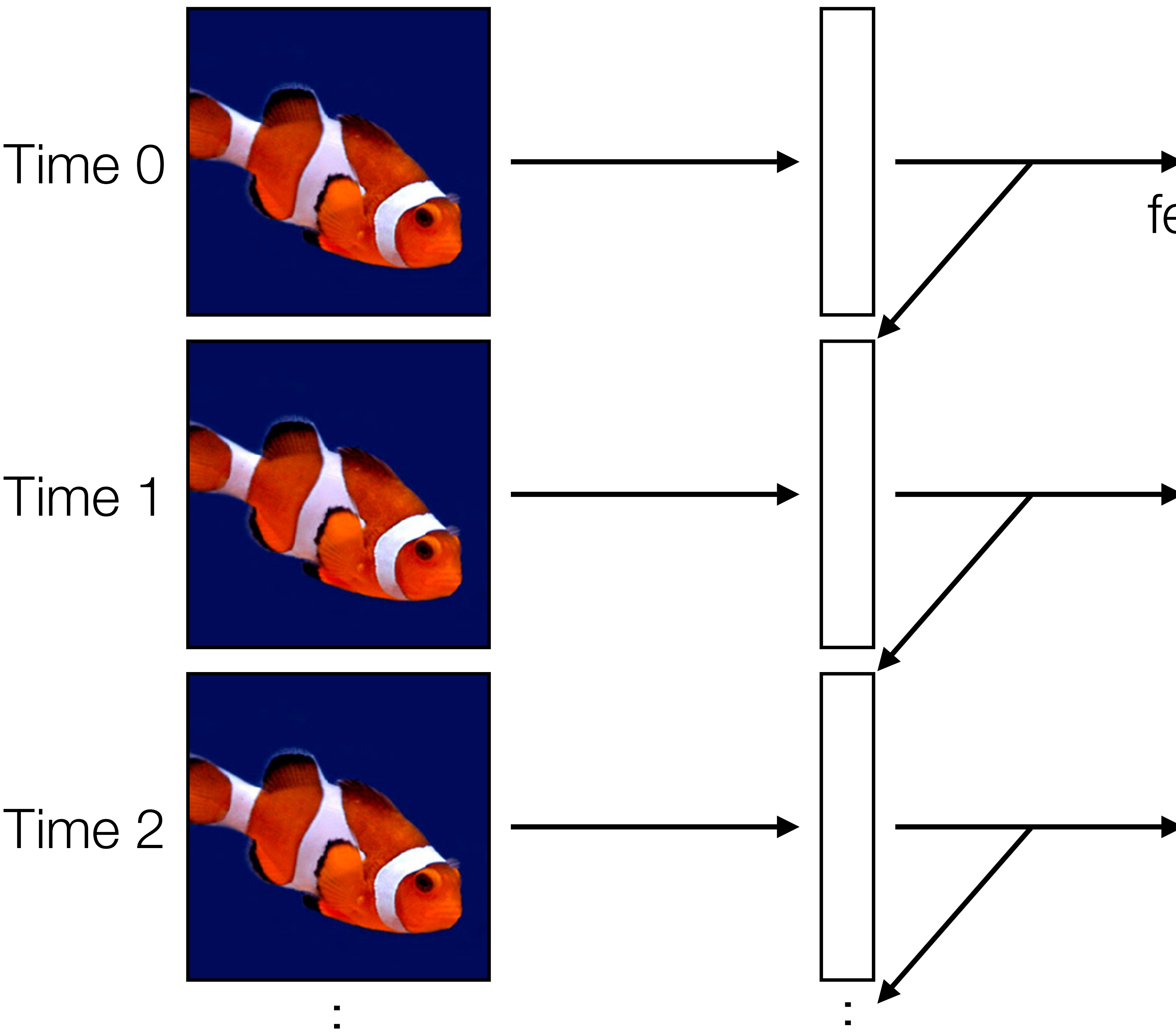
What if we had a video?

# RNNs

Feedforward NN's:
treat each video frame
separately



# RNNs

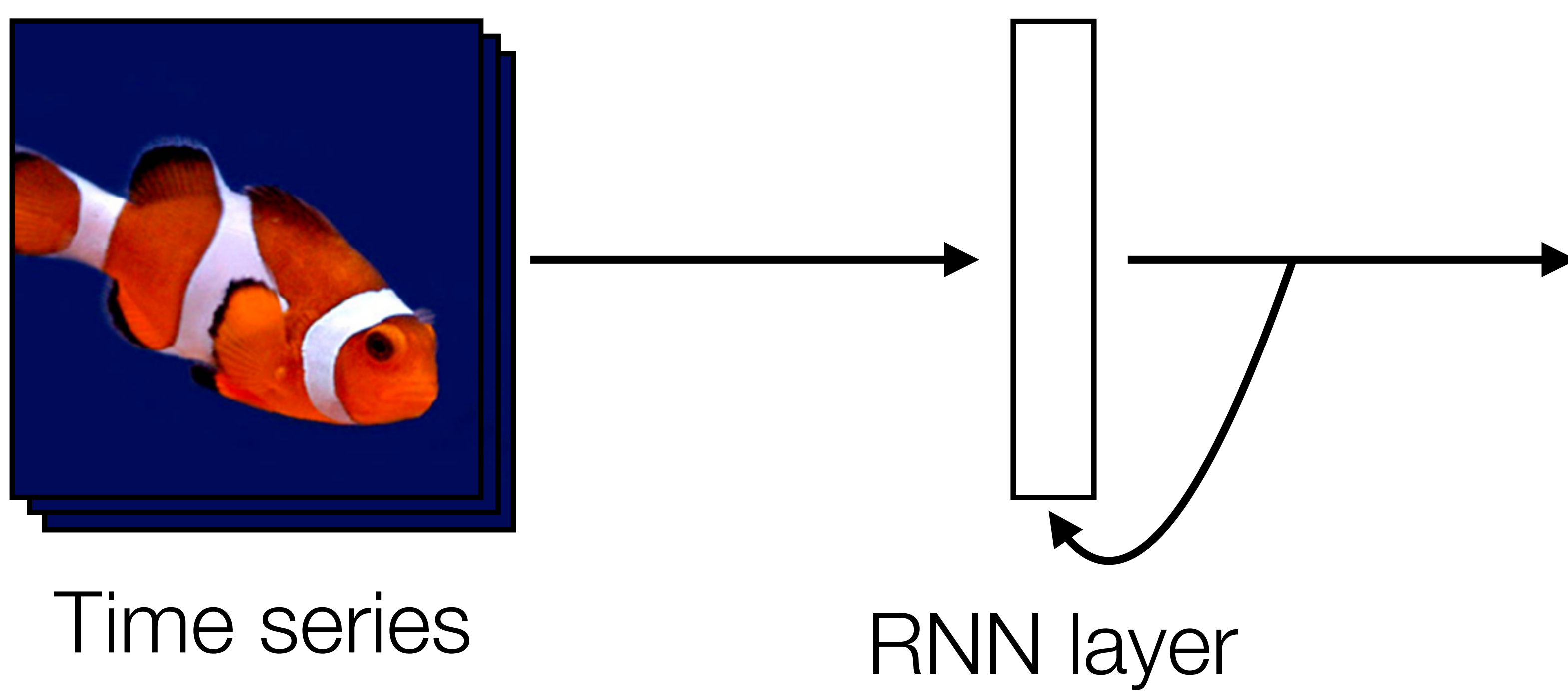



Feedforward NN's:
treat each video frame separately

RNNs:
feed output at previous time step as input to RNN layer at current time step

In `keras`, different RNN options:
`SimpleRNN`, `LSTM`, `GRU`

# RNNs




Feedforward NN's:
treat each video frame separately

RNNs:
feed output at previous time step as input to RNN layer at current time step

In `keras`, different RNN options: `SimpleRNN`, `LSTM`, `GRU`

# Example: SimpleRNN

memory stored in `current_state` variable!

```
current_state = np.zeros(num_neurons)

for input in input_sequence:

  output = activation(np.dot(input, W)
                      + np.dot(current_state, U)
                      + b)

  current_state = output
```

Activation function could, for instance, be ReLU

Parameters: weight matrices `W` & `U`, and bias vector `b`

Key idea: **it's like a dense layer in a `for` loop with some memory!**

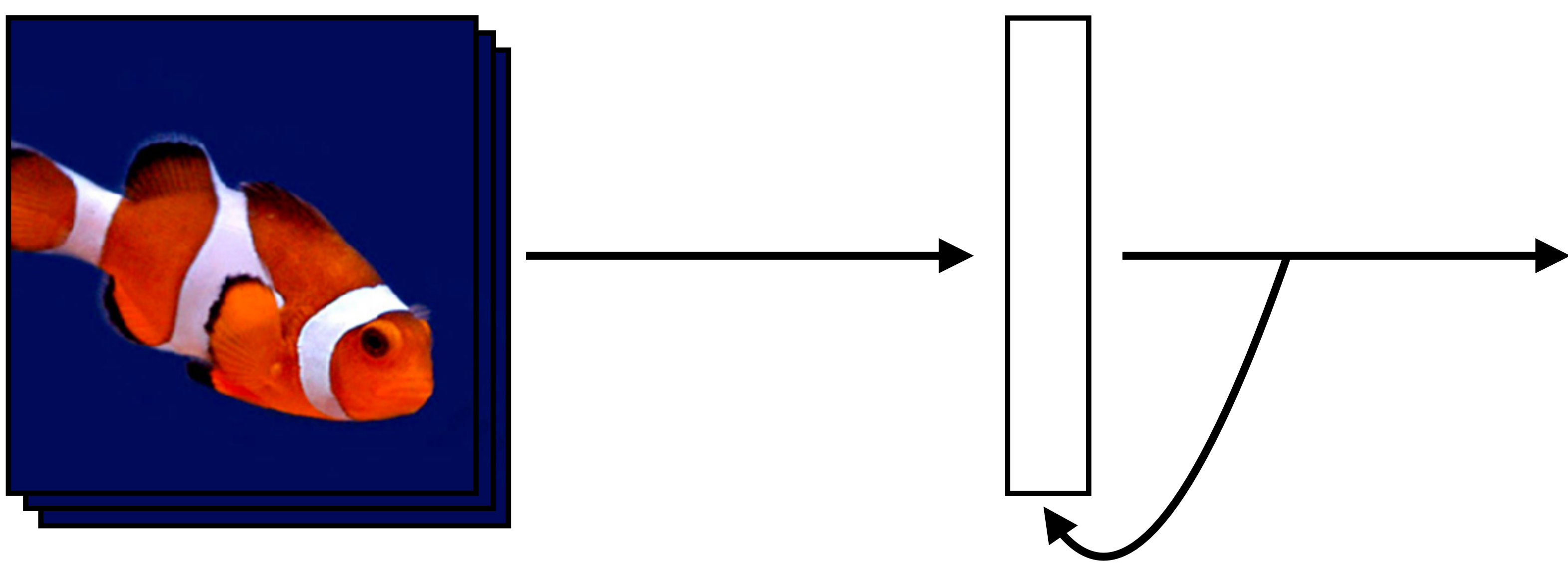
# RNNs

Feedforward NN's:
treat each video frame separately

RNNs:
feed output at previous time step as input to RNN layer at current time step

In `keras`, different RNN options:
`SimpleRNN`, `LSTM`, `GRU`

readily chains together with other neural net layers



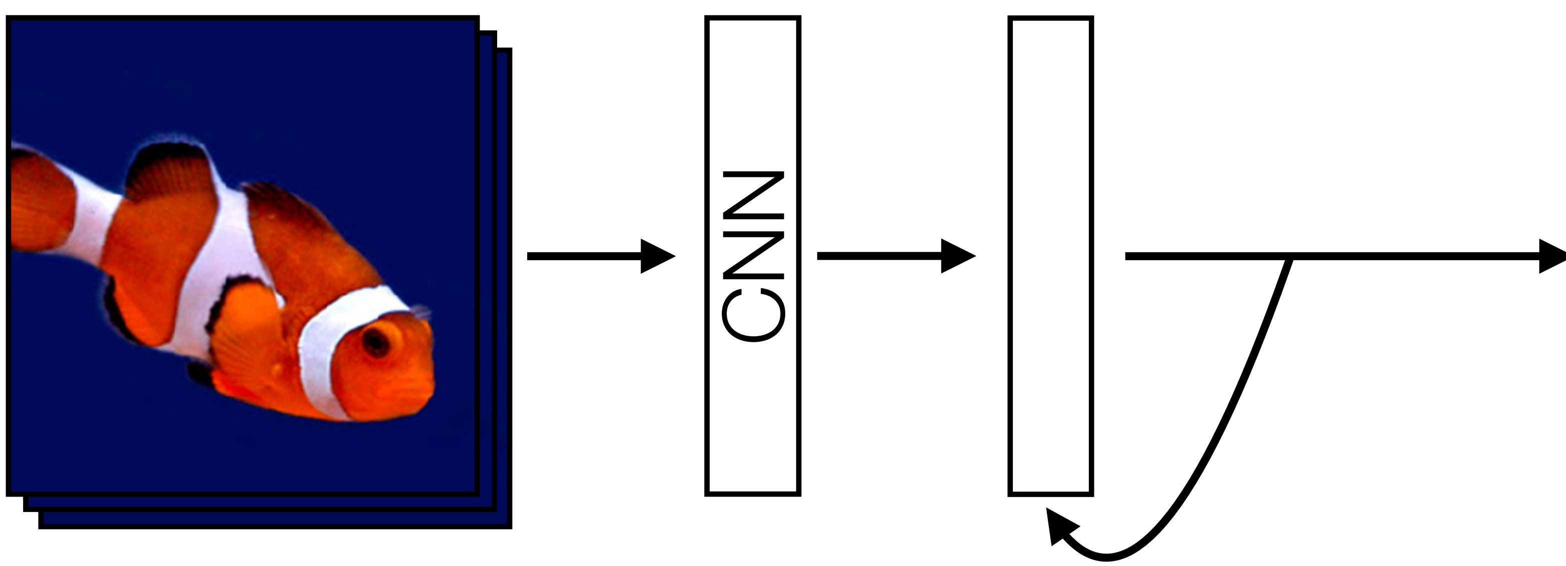
Time series

RNN layer

like a dense layer that has memory

# RNNs

readily chains together with other neural net layers




Time series

RNN layer

like a dense layer that has memory

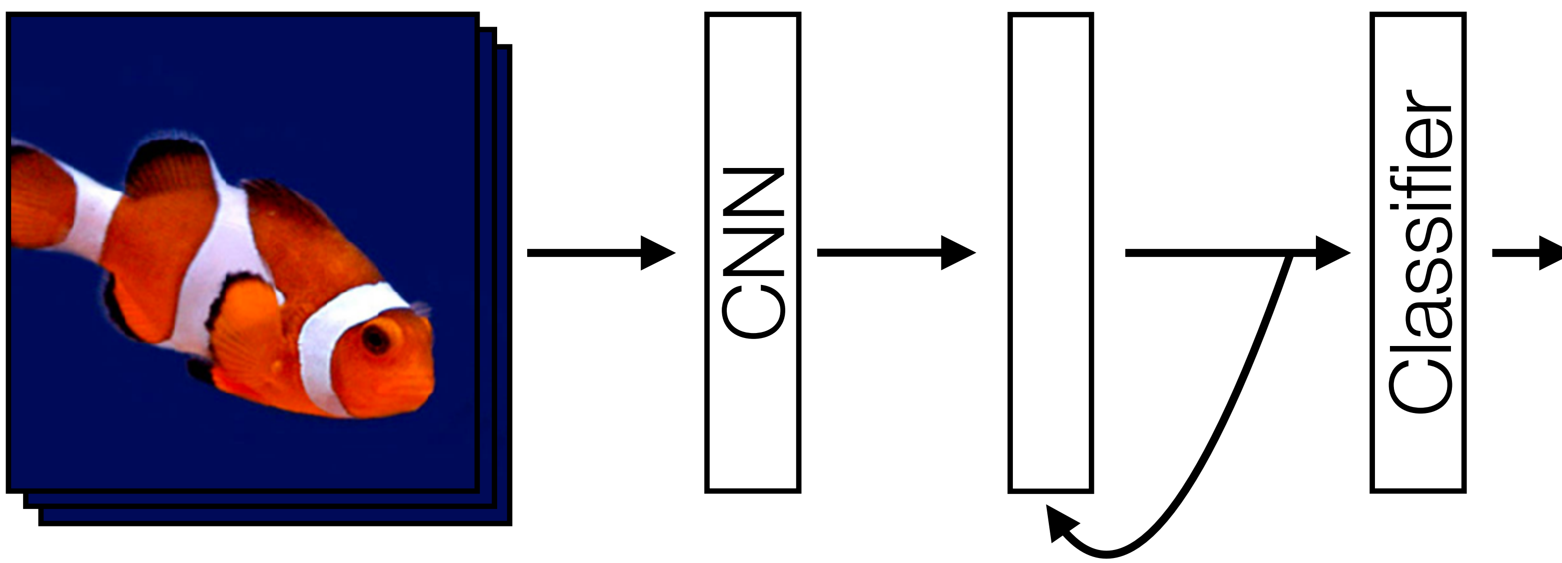
Feedforward NN's: treat each video frame separately

RNNs: feed output at previous time step as input to RNN layer at current time step

In `keras`, different RNN options: `SimpleRNN`, `LSTM`, `GRU`

# RNNs

readily chains together with
other neural net layers




Time series

RNN layer

like a dense layer
that has memory

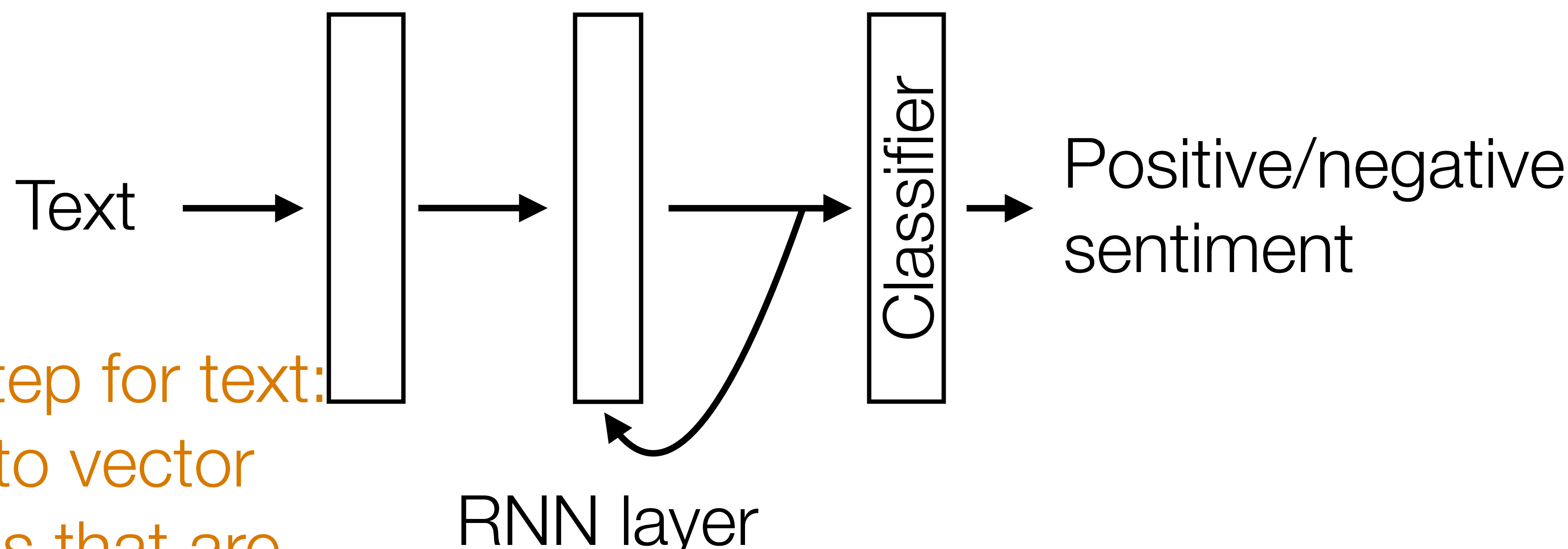
Feedforward NN's:
treat each video frame
separately

RNNs:
feed output at previous
time step as input to
RNN layer at current
time step

In `keras`, different
RNN options:
`SimpleRNN`, `LSTM`,
`GRU`

# RNNs

Example: Given text (e.g., movie review, Tweet), figure out whether it has positive or negative sentiment (binary classification)




Common first step for text: turn words into vector representations that are semantically meaningful

# (Flashback) Do Data Actually Live on Manifolds?

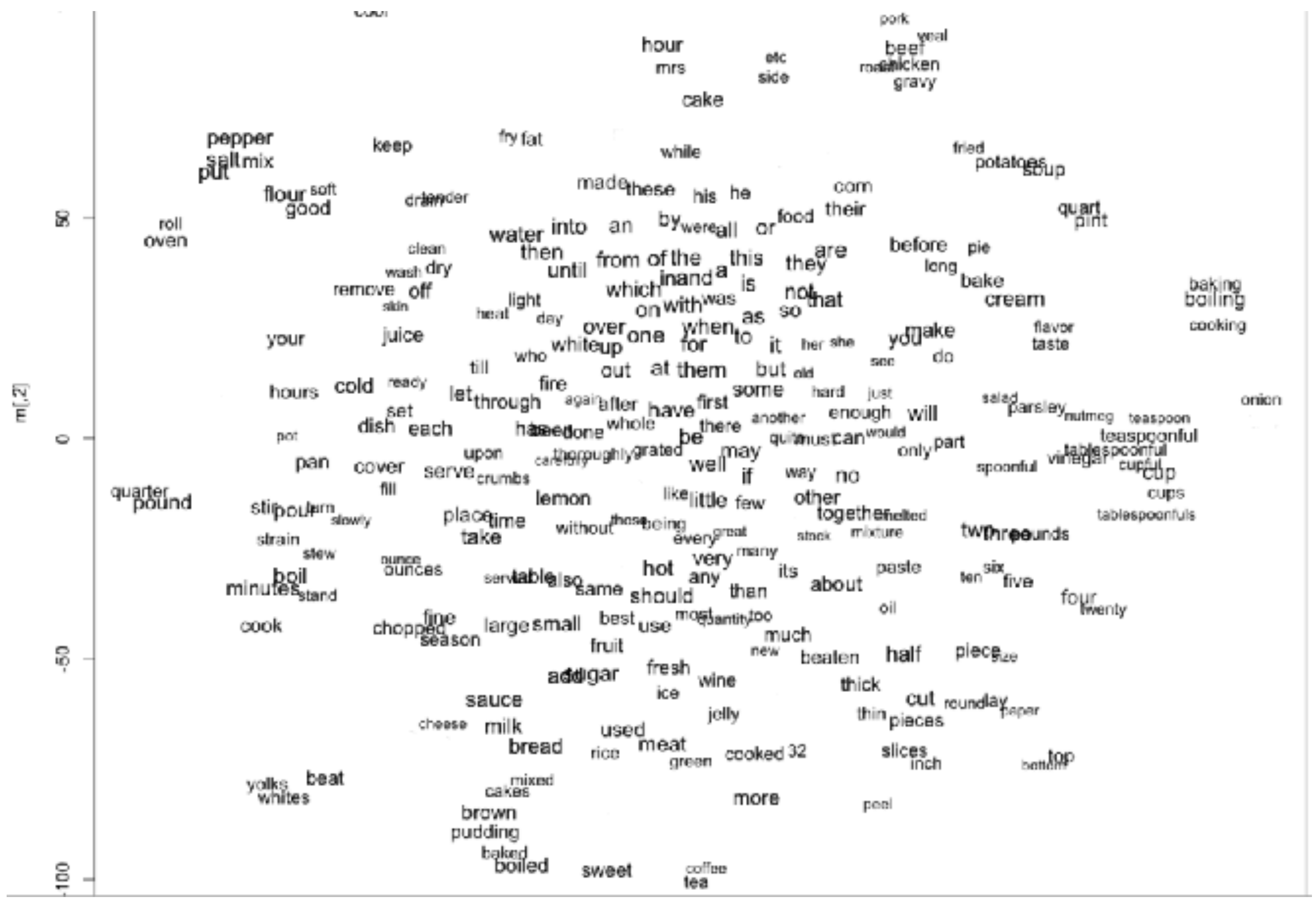
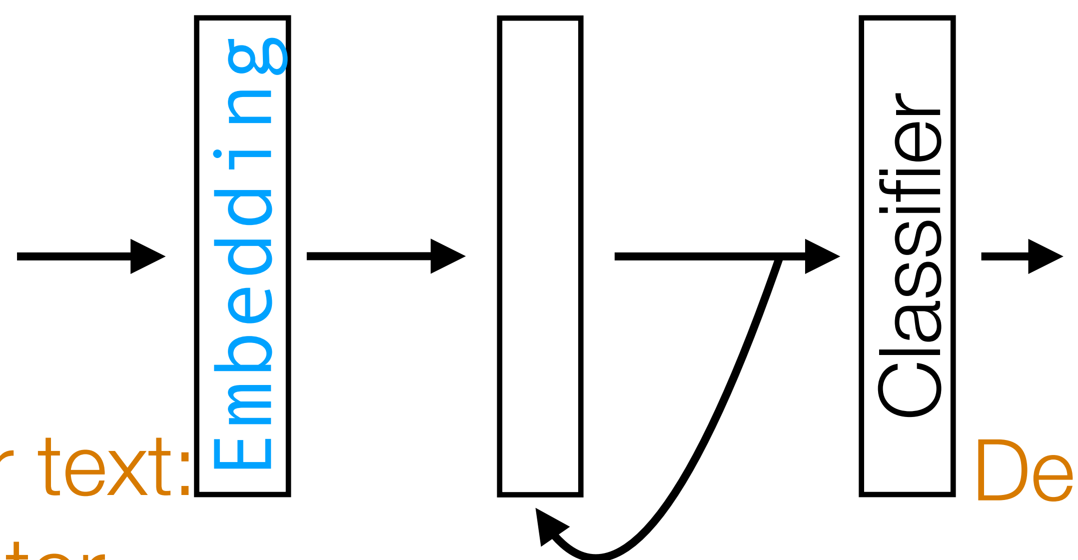



Image source: http://www.adityathakker.com/wp-content/uploads/2017/06/word-embeddings-994x675.png

# RNNs

Example: Given text (e.g., movie review, Tweet), figure out whether it has positive or negative sentiment (binary classification)




Common first step for text: turn words into vector representations that are semantically meaningful

In `keras`, use the `Embedding` layer

Dense layer, 2 neurons, softmax activation

# RNNs

Demo

# RNNs: a little bit more detail

# (Flashback) Example: SimpleRNN

memory stored in `current_state` variable!

```
current_state = np.zeros(num_neurons)

for input in input_sequence:

    output = activation(np.dot(input, W)
                        + np.dot(current_state, U)
                        + b)

    current_state = output
```

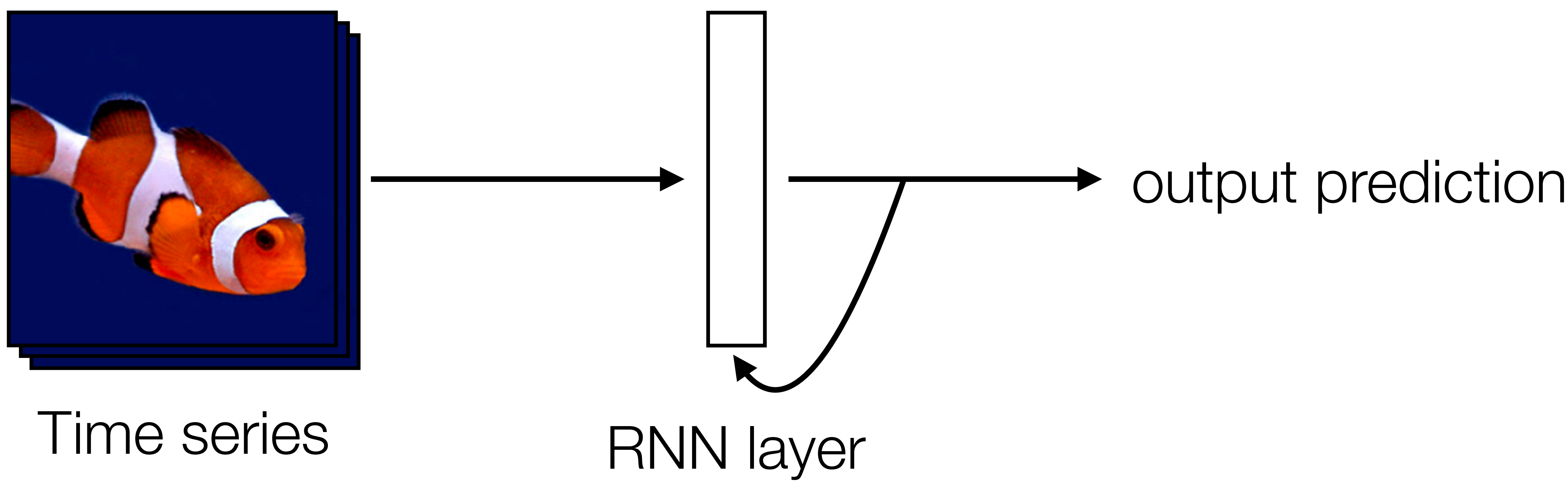
Activation function could, for instance, be ReLU

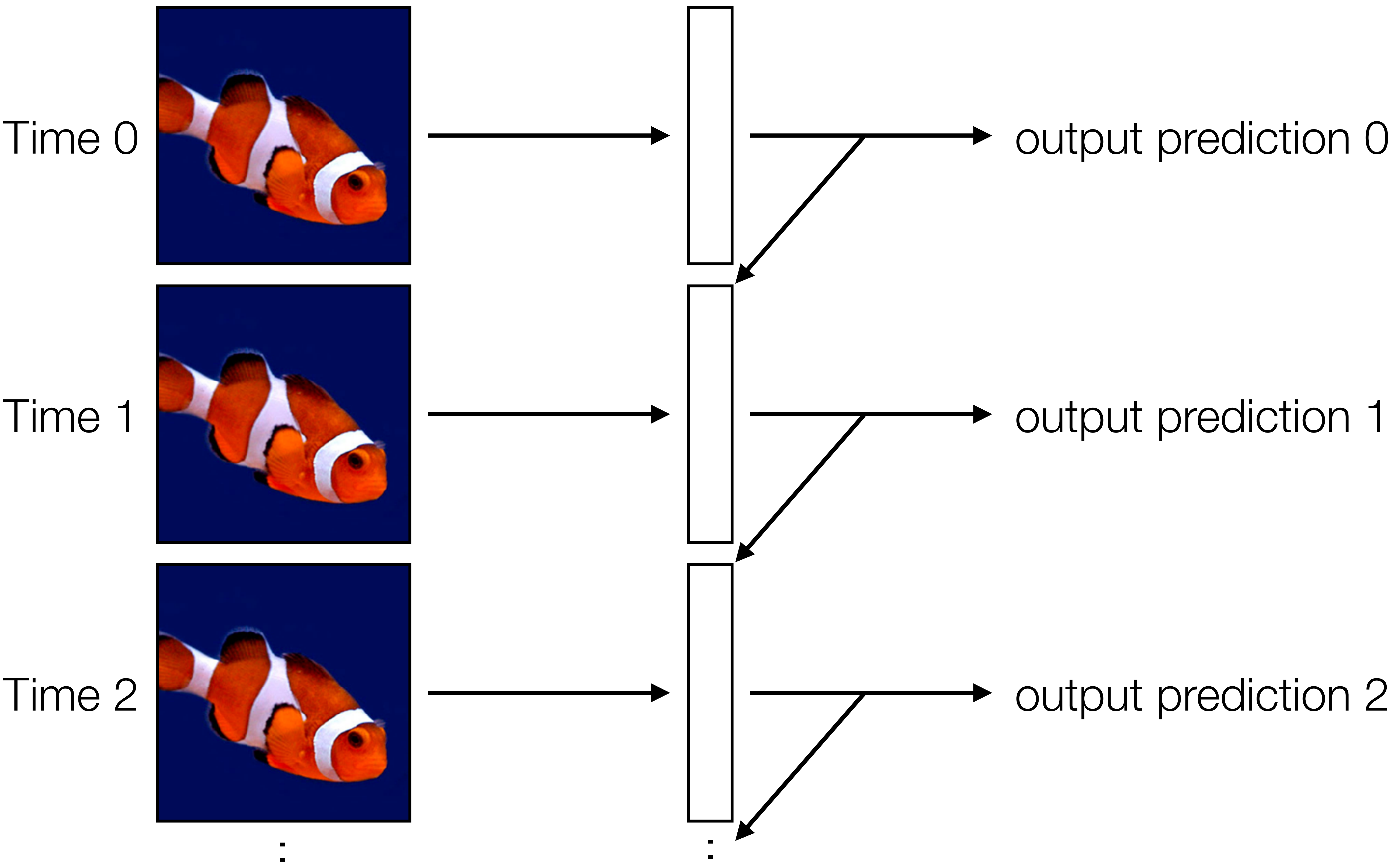
Parameters: weight matrices `W` & `U`, and bias vector `b`

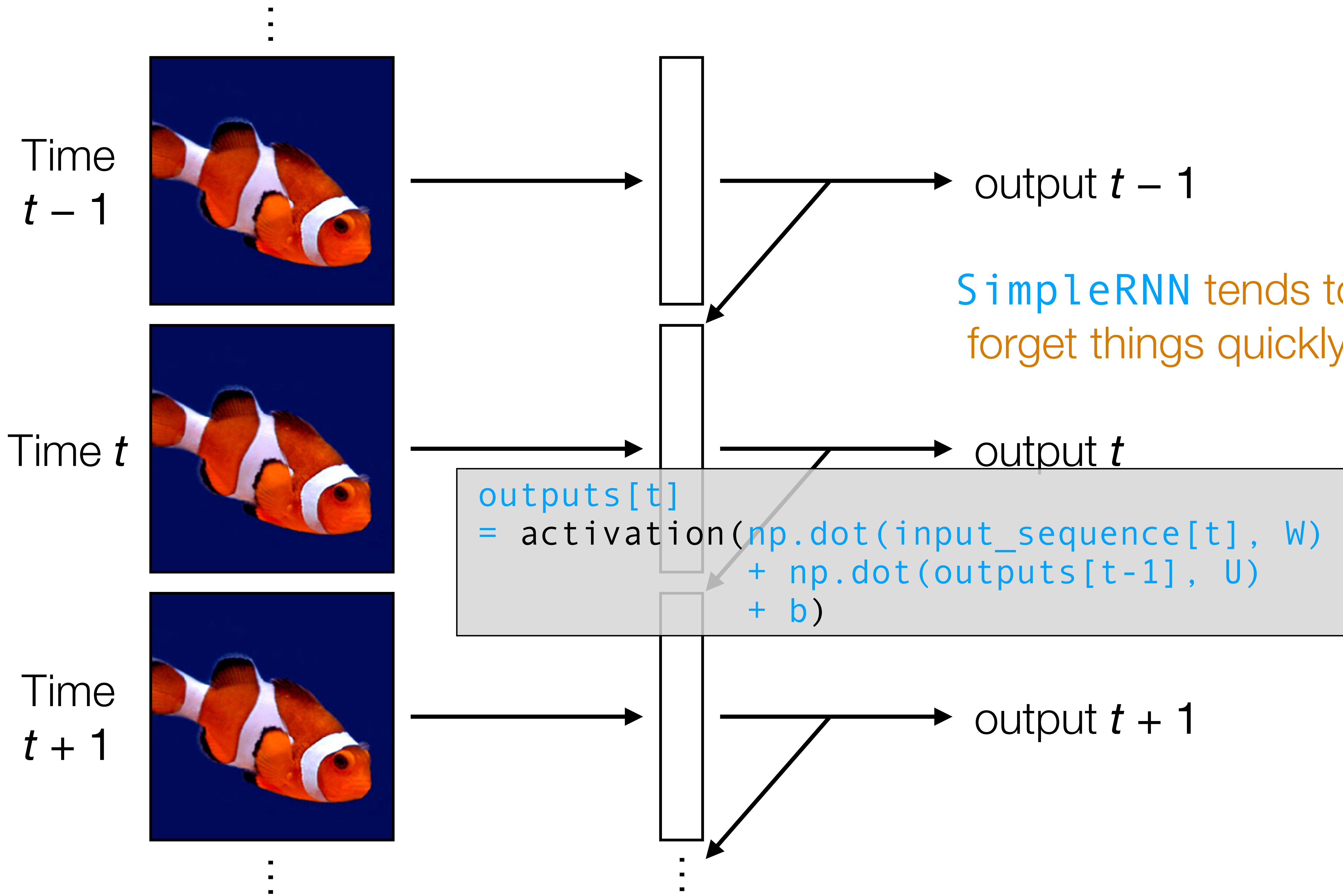
Key idea: **it's like a dense layer in a `for` loop with some memory!**

memory stored in `current_state` variable!

```
current_state = np.zeros(num_neurons)

outputs = []

for input in input_sequence:

  output = activation(np.dot(input, W)
                      + np.dot(current_state, U)
                      + b)

  current_state = output

  outputs.append(output)
```

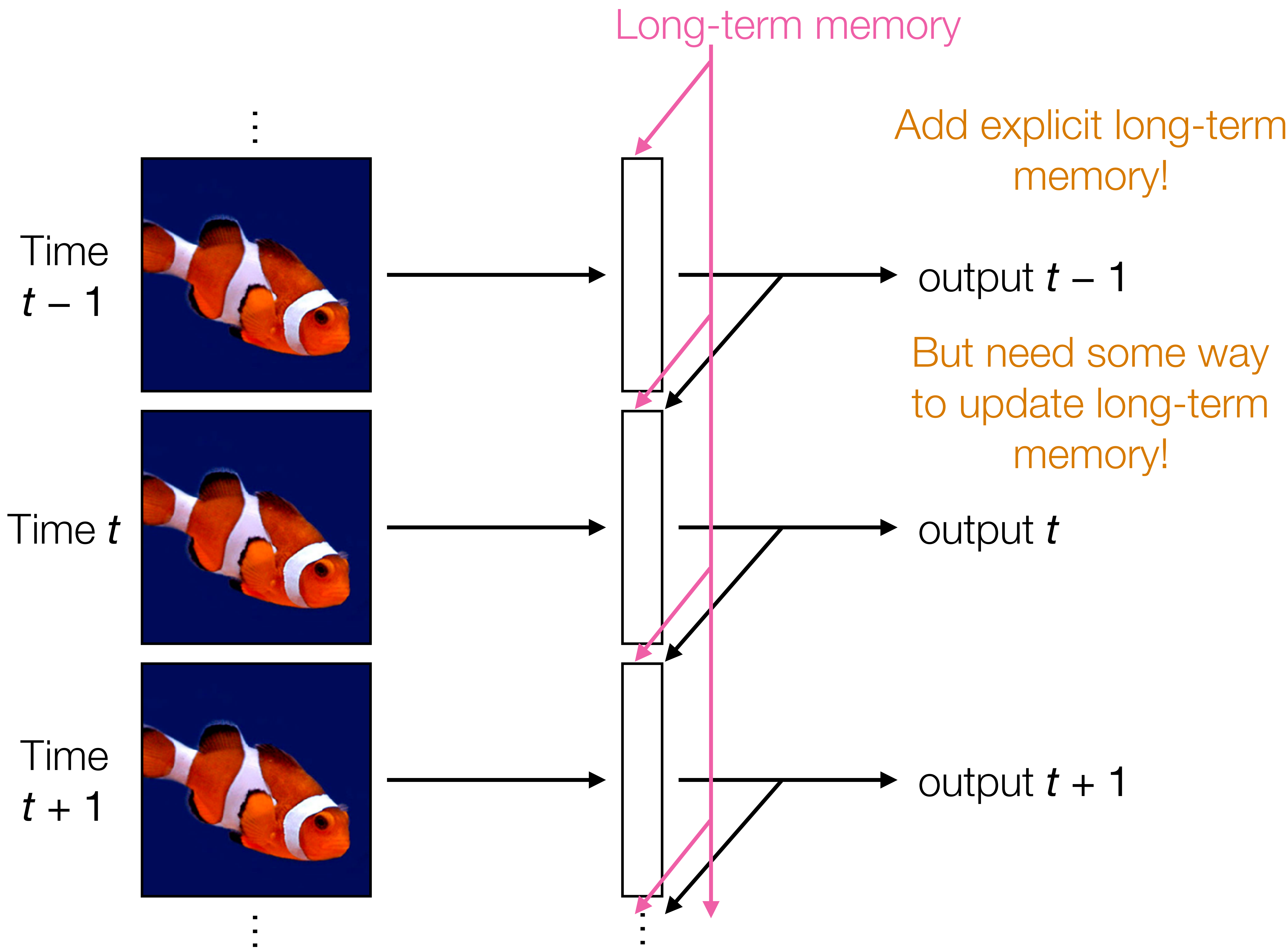

Time series
RNN layer
output prediction

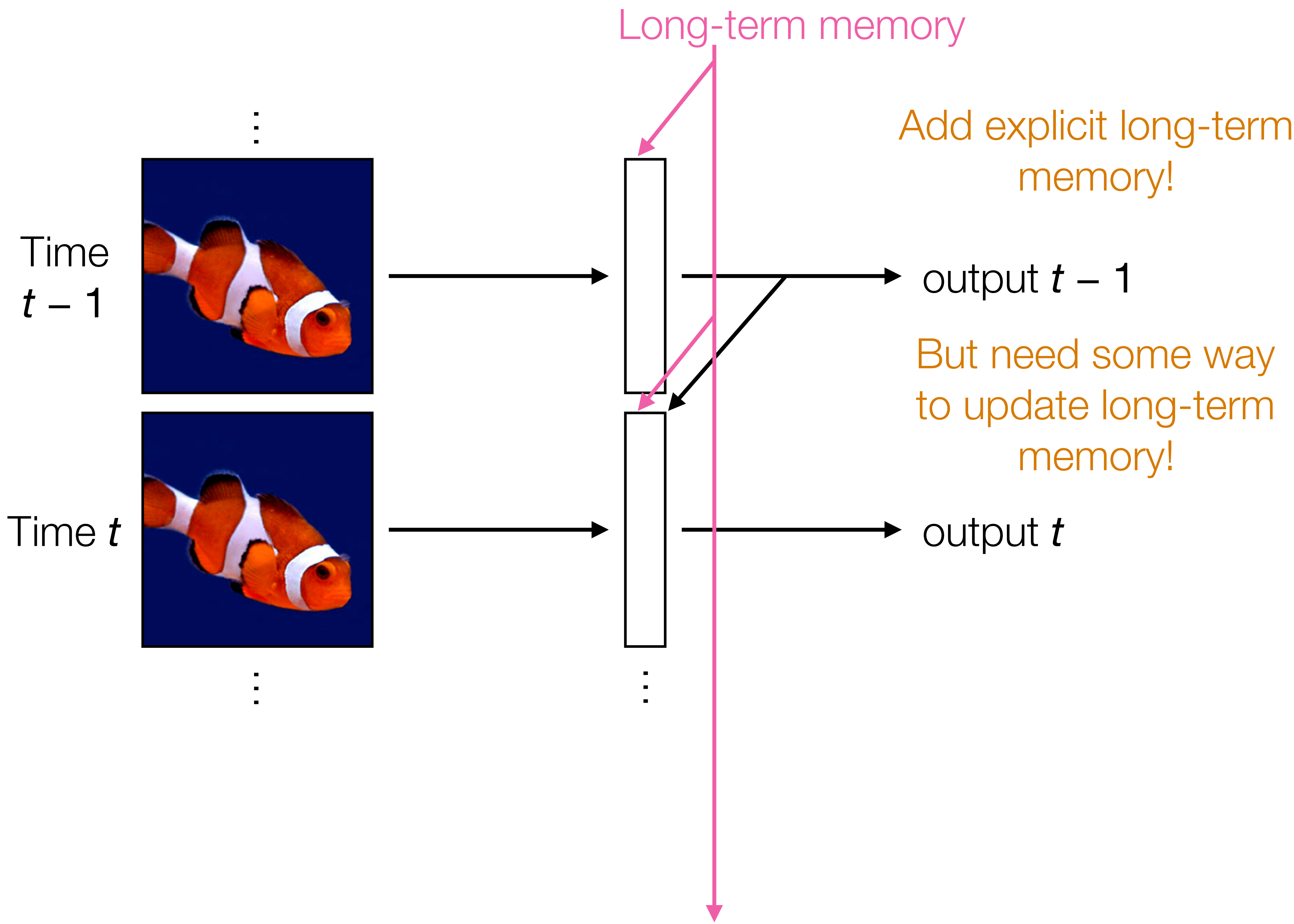

Time 0
output prediction 0
Time 1
output prediction 1
Time 2
output prediction 2



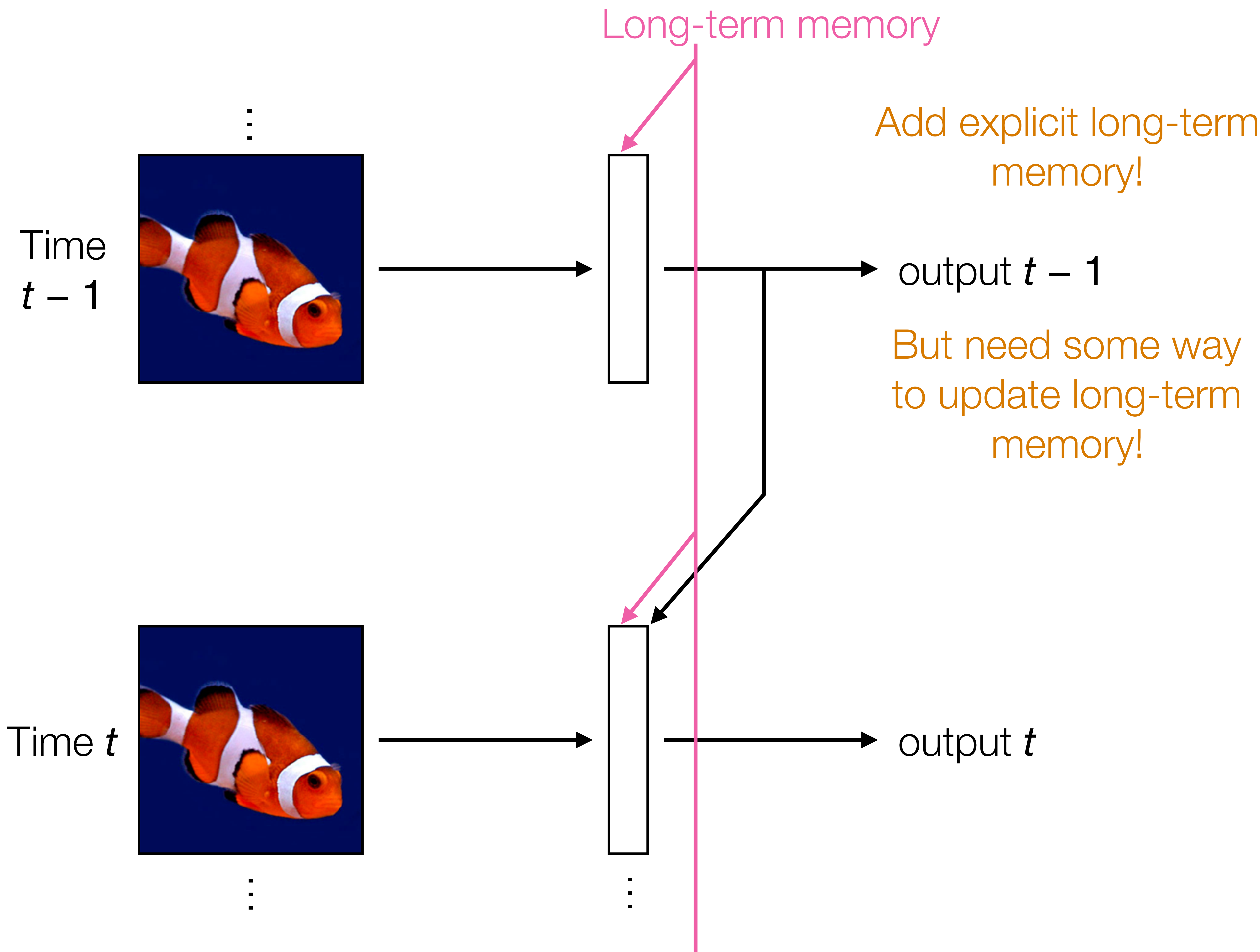

SimpleRNN tends to forget things quickly

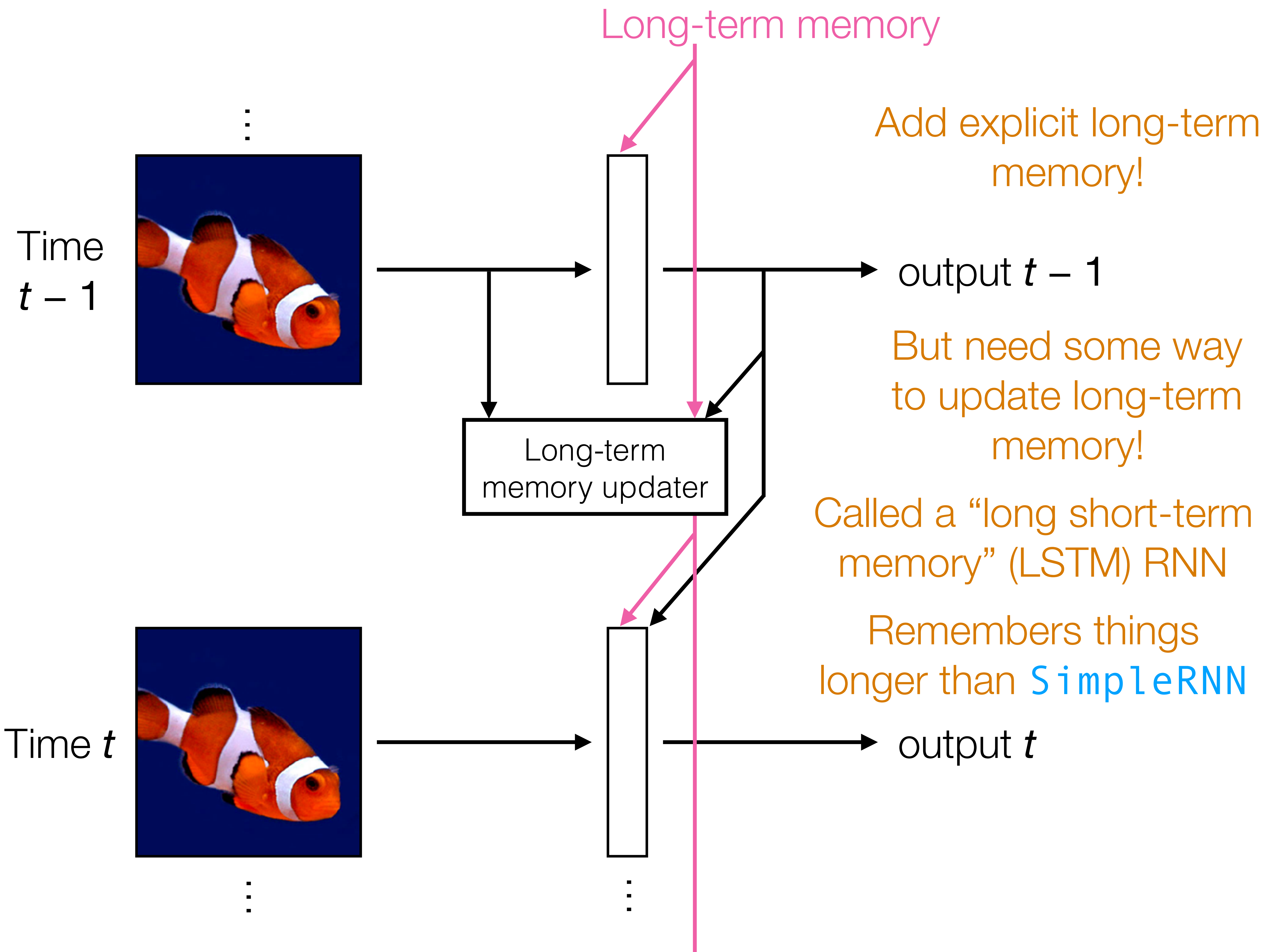
```
outputs[t]
= activation(np.dot(input_sequence[t], W)
             + np.dot(outputs[t-1], U)
             + b)
```


Long-term memory
Add explicit long-term memory!
Time $t-1$
output $t-1$
But need some way to update long-term memory!
Time $t$
output $t$
Time $t+1$
output $t+1$



Long-term memory
Add explicit long-term memory!
Time $t-1$
output $t-1$
But need some way to update long-term memory!
Time $t$
output $t$


Long-term memory
Add explicit long-term memory!
Time $t-1$
output $t-1$
But need some way to update long-term memory!
Time $t$
output $t$


Long-term memory
Time
t − 1
output t − 1
Long-term
memory updater
Time t
output t
Add explicit long-term
memory!
But need some way
to update long-term
memory!
Called a “long short-term
memory” (LSTM) RNN
Remembers things
longer than SimpleRNN

# RNNs

- Neatly handles time series, remembering things over time

- An RNN layer by itself doesn't take advantage of image/text structure!
  - For images: combine with CNN basic building block (convolutional layer + pooling)
  - For text: combine with embedding layer

# Analyzing Times Series with CNNs

- Think about an image with 1 column, and where the rows index time steps: this is a time series!

- Think about a 2D image where rows index time steps, and the columns index features: this is a multivariate time series (feature vector that changes over time!)

- CNNs can be used to analyze time series *but inherently the size of the filters used say how far back in time we look*

- If your time series does not have long-range dependencies that require long-term memory, CNNs can do well already!

    - If you need long-term memory, use RNNs

# Some other deep learning topics

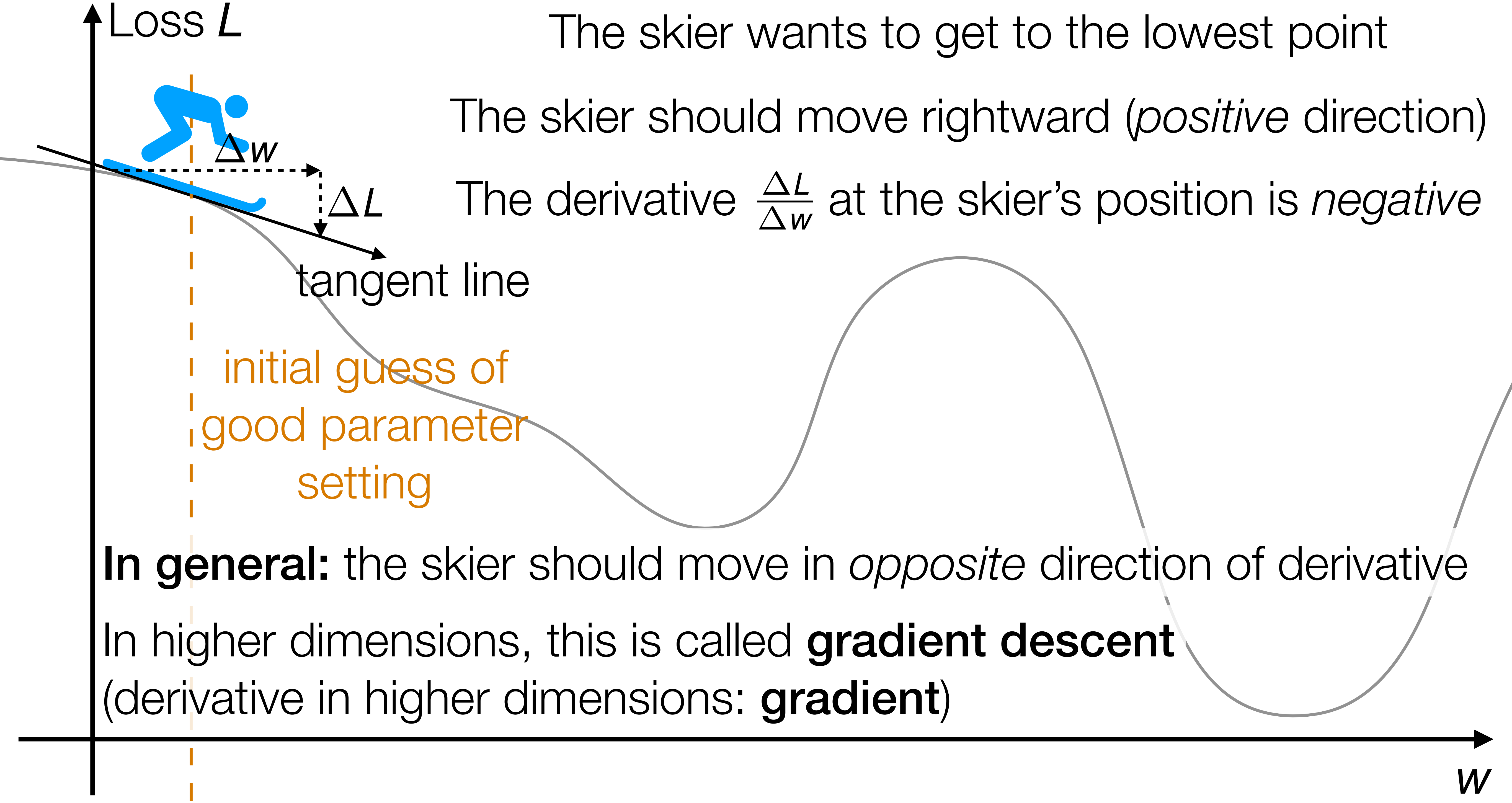
# Learning a Deep Net

Suppose the neural network has a single real number parameter *w*

The skier wants to get to the lowest point

The skier should move rightward (*positive* direction)

The derivative $\frac{\Delta L}{\Delta w}$ at the skier's position is *negative*

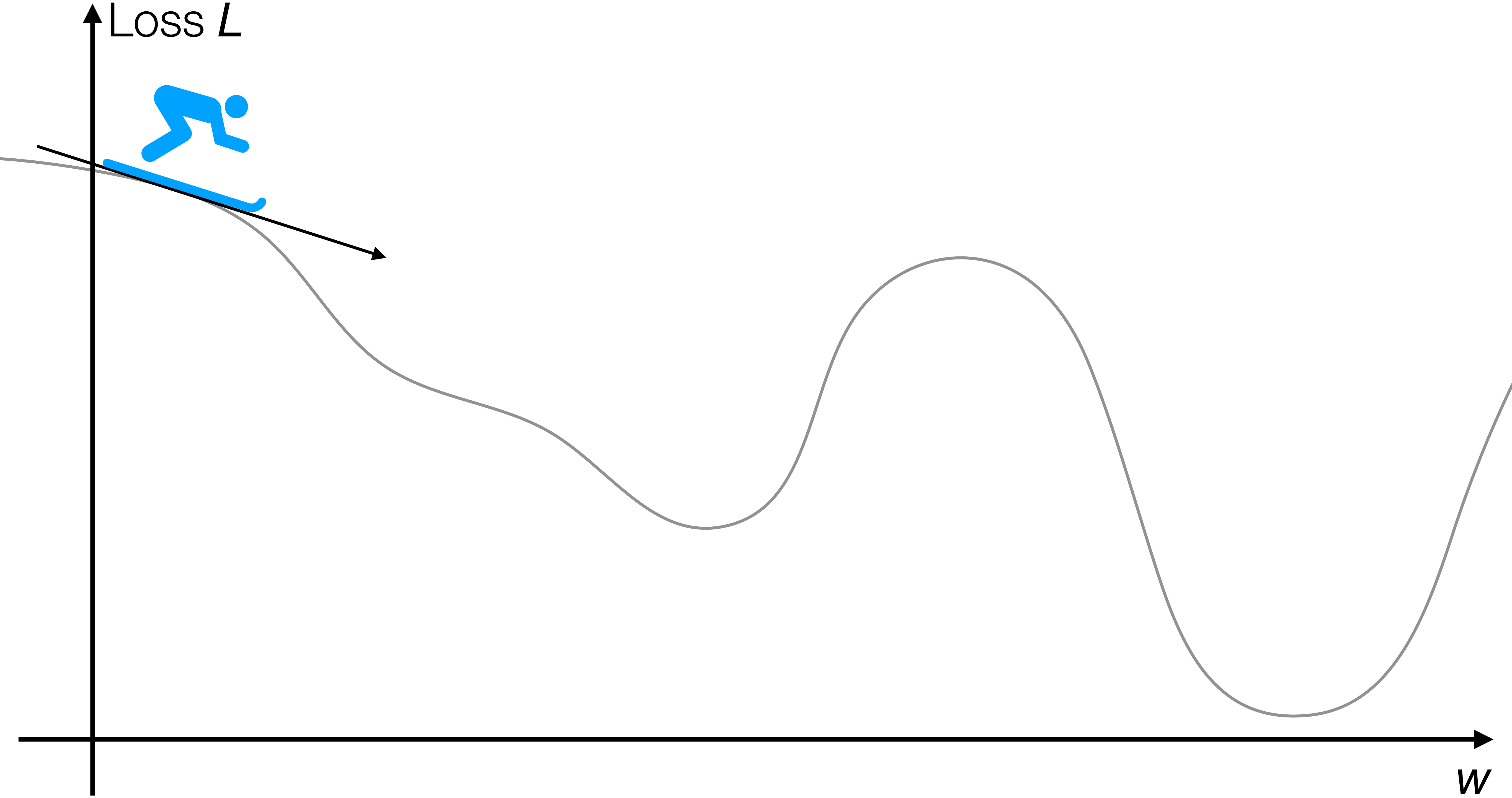



**In general:** the skier should move in *opposite* direction of derivative

In higher dimensions, this is called **gradient descent** (derivative in higher dimensions: **gradient**)

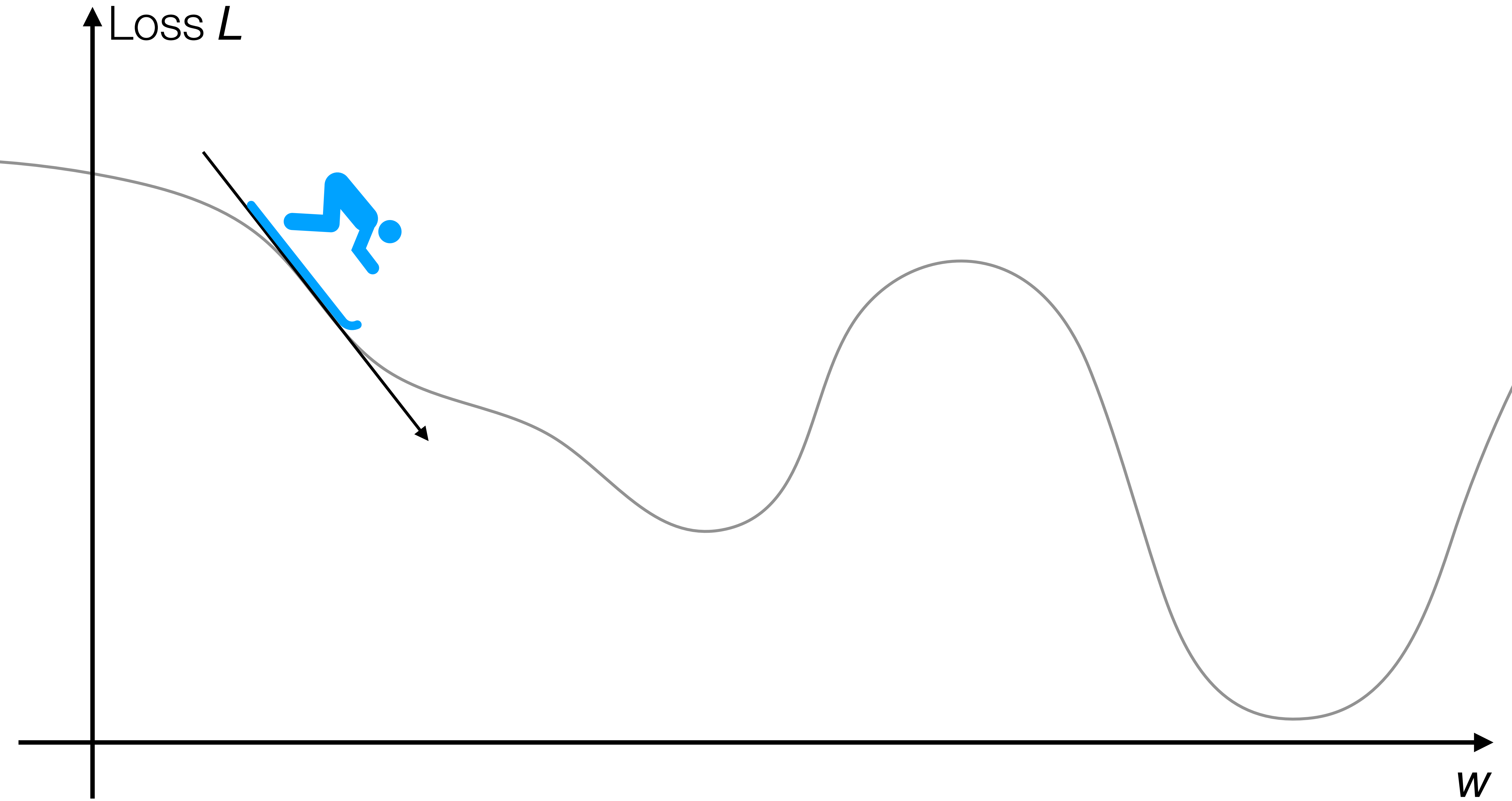
# Learning a Deep Net

Suppose the neural network has a single real number parameter ***w***



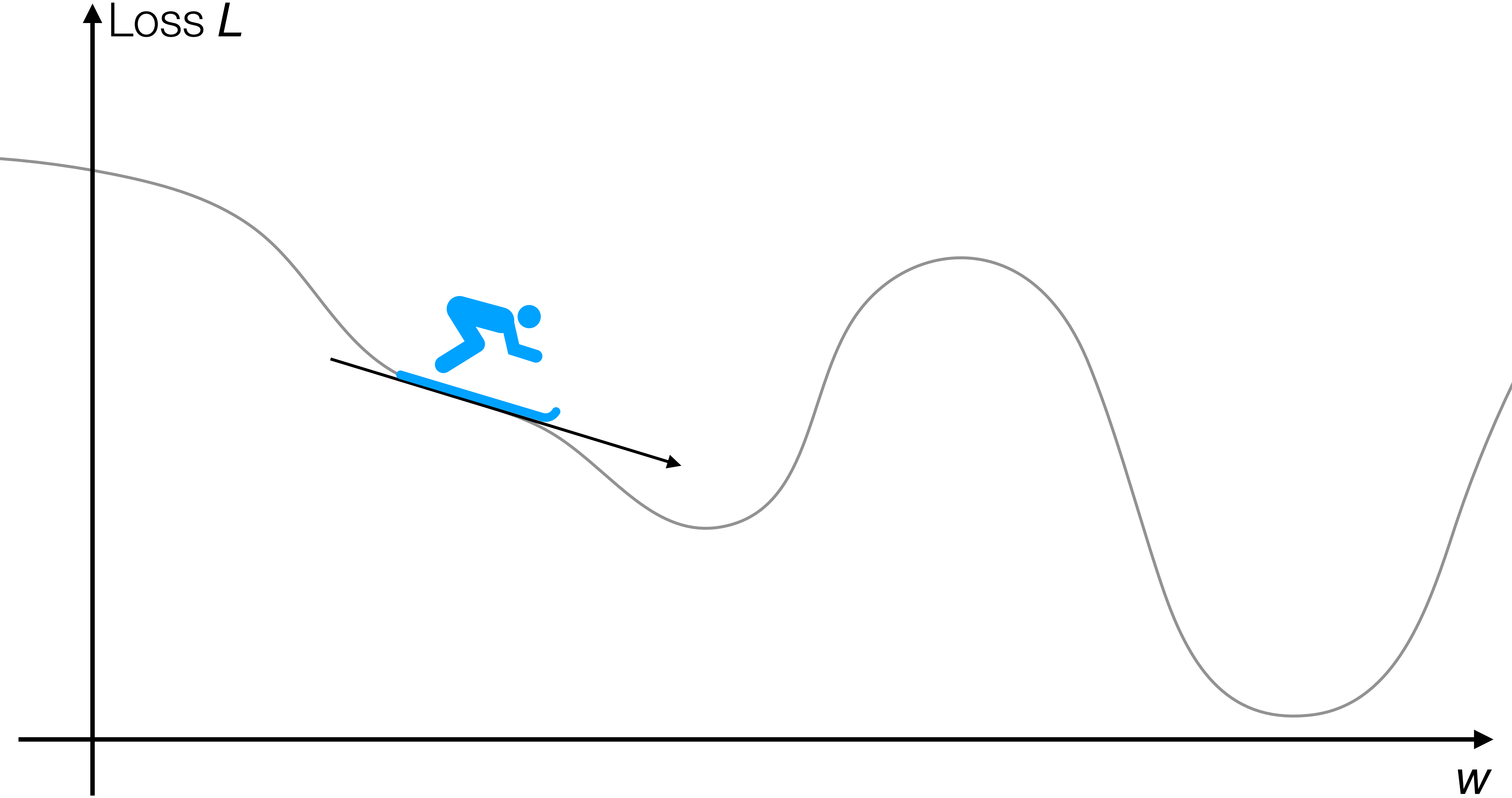
# Learning a Deep Net

Suppose the neural network has a single real number parameter *w*



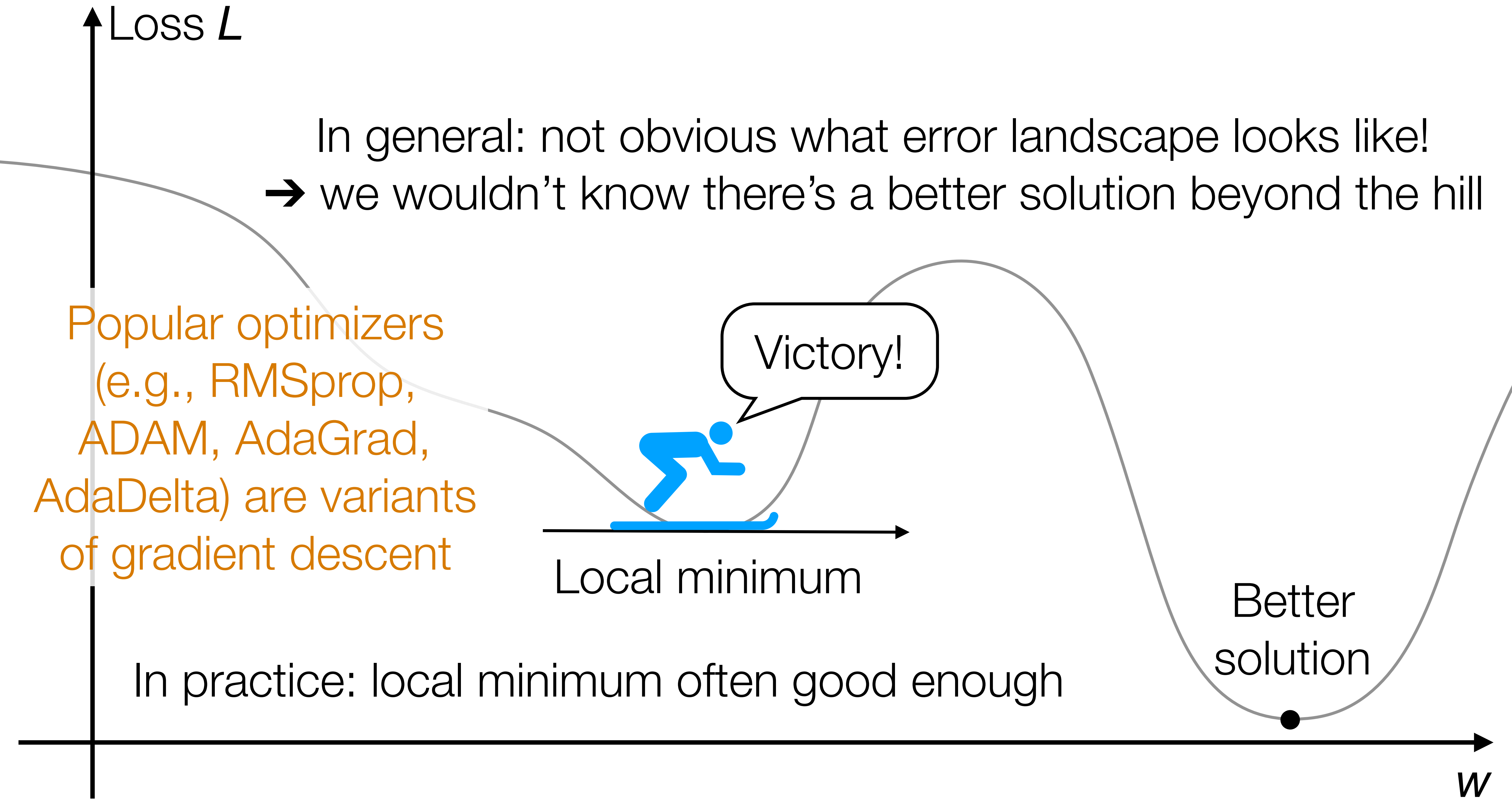
# Learning a Deep Net

Suppose the neural network has a single real number parameter $w$



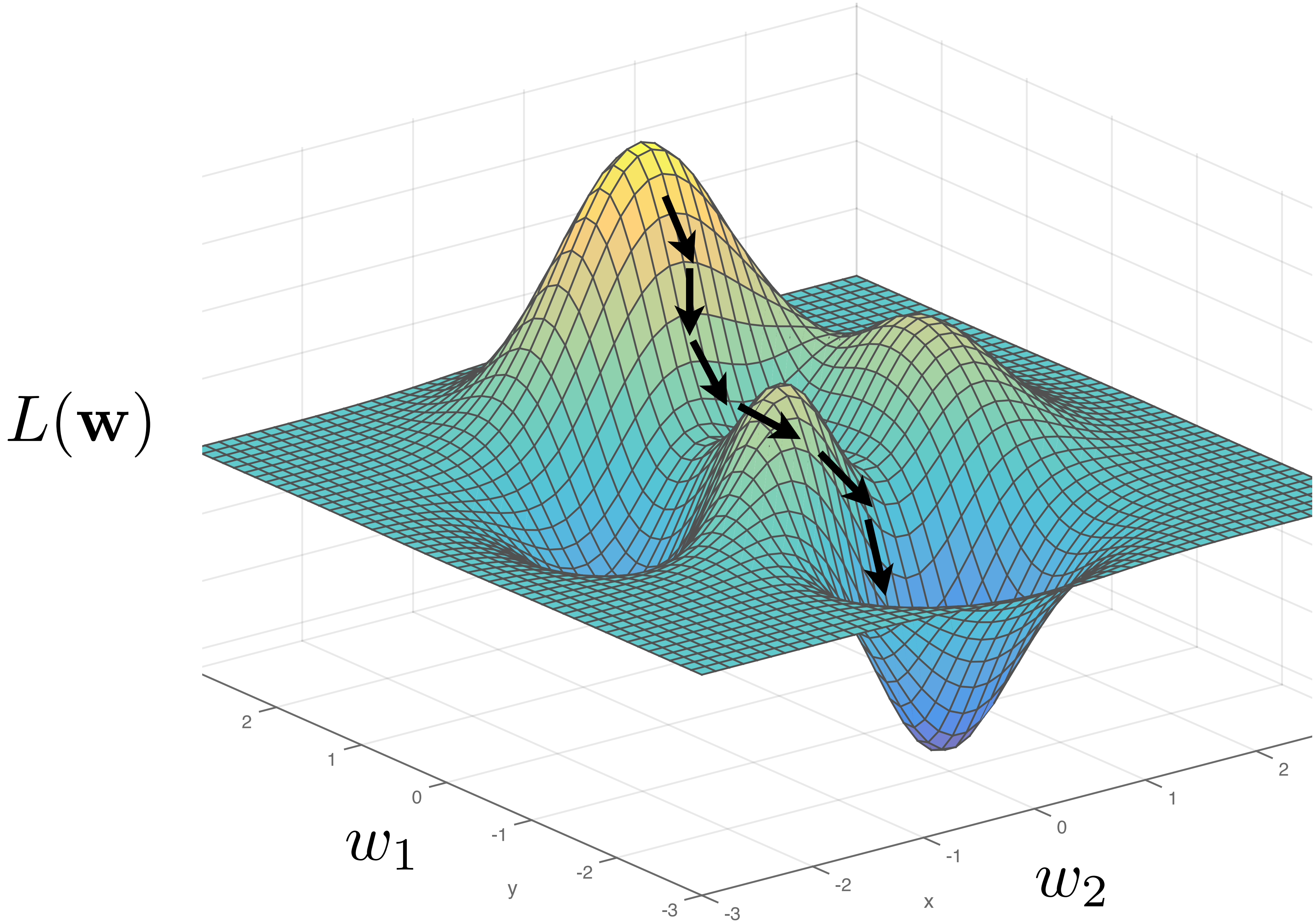
# Learning a Deep Net

Suppose the neural network has a single real number parameter *w*



# Learning a Deep Net

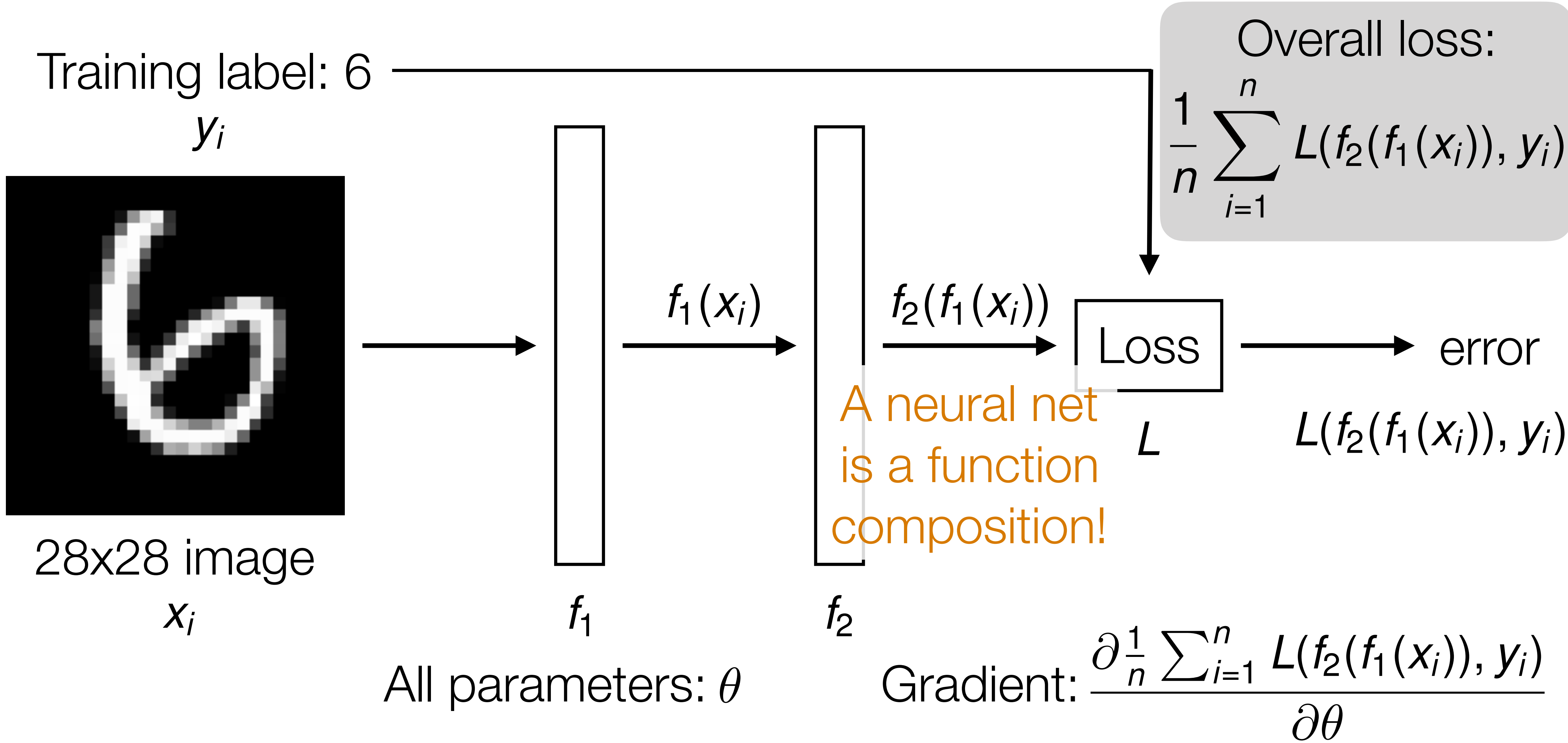
2D example




Slide by Phillip Isola

Remark: In practice, deep nets often have > *millions* of parameters, so *very* high-dimensional gradient descent

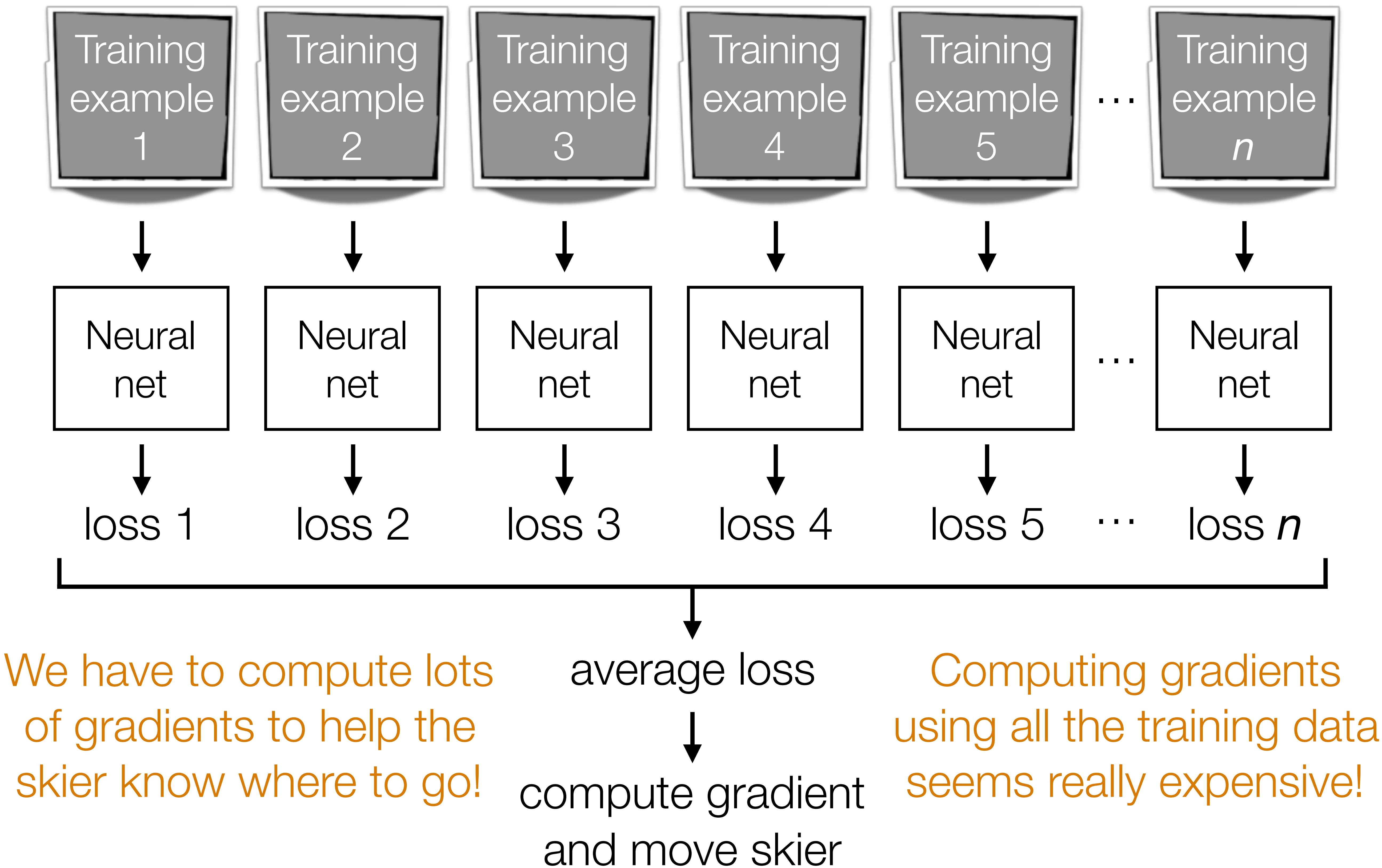
# Handwritten Digit Recognition



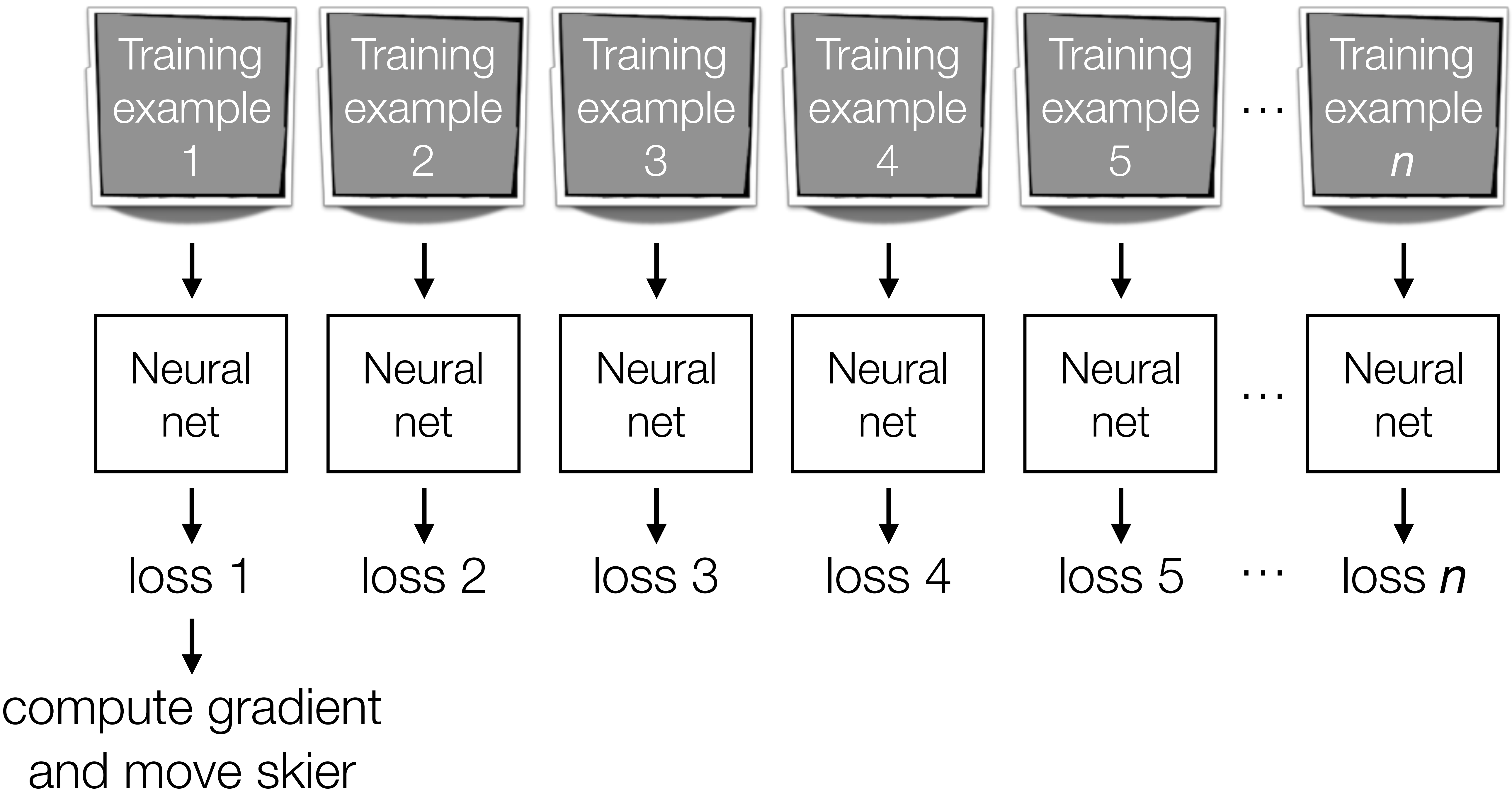

**Automatic differentiation** is crucial in learning deep nets!

Careful derivative chain rule calculation: **back-propagation**

# Gradient Descent

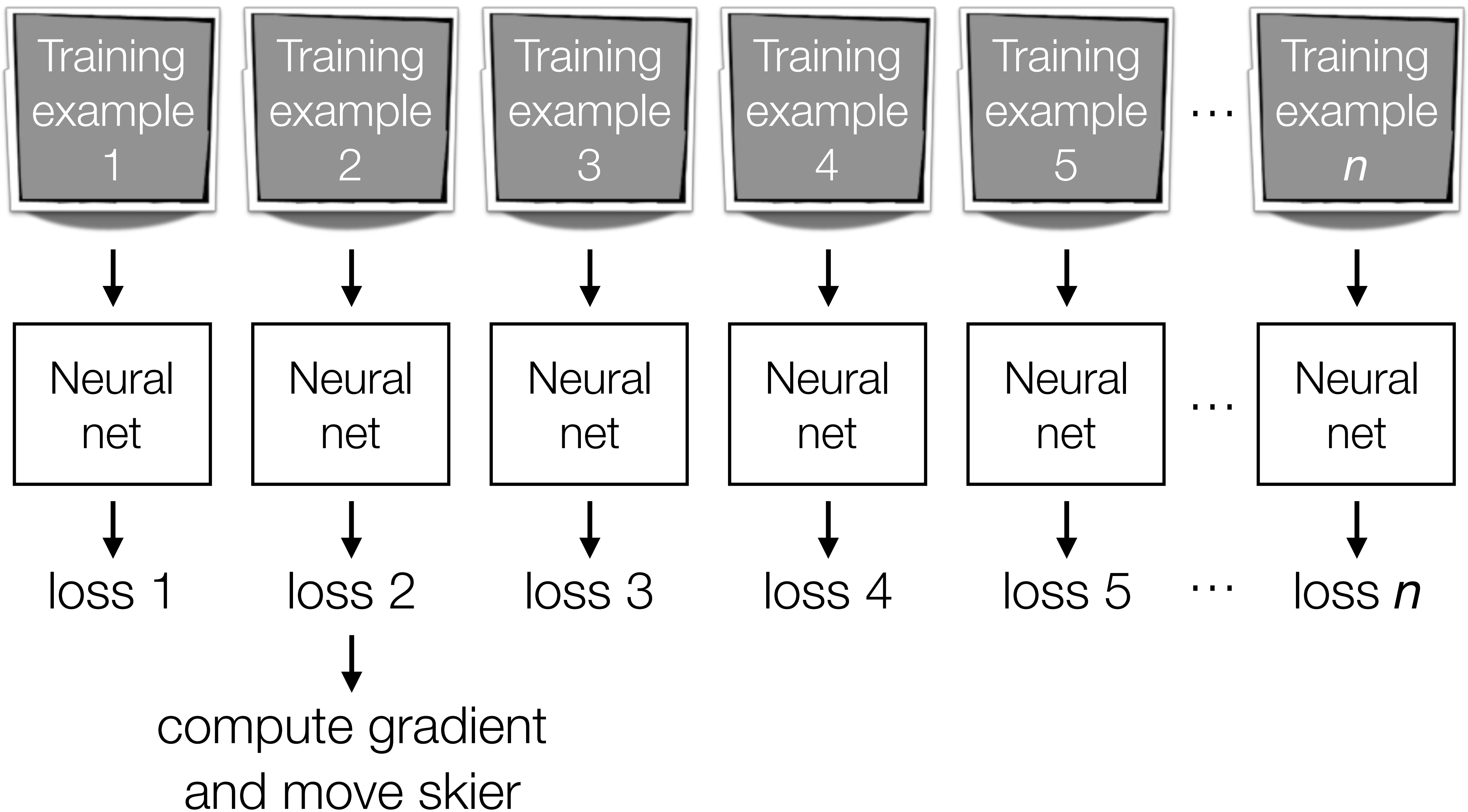


# Stochastic Gradient Descent (SGD)



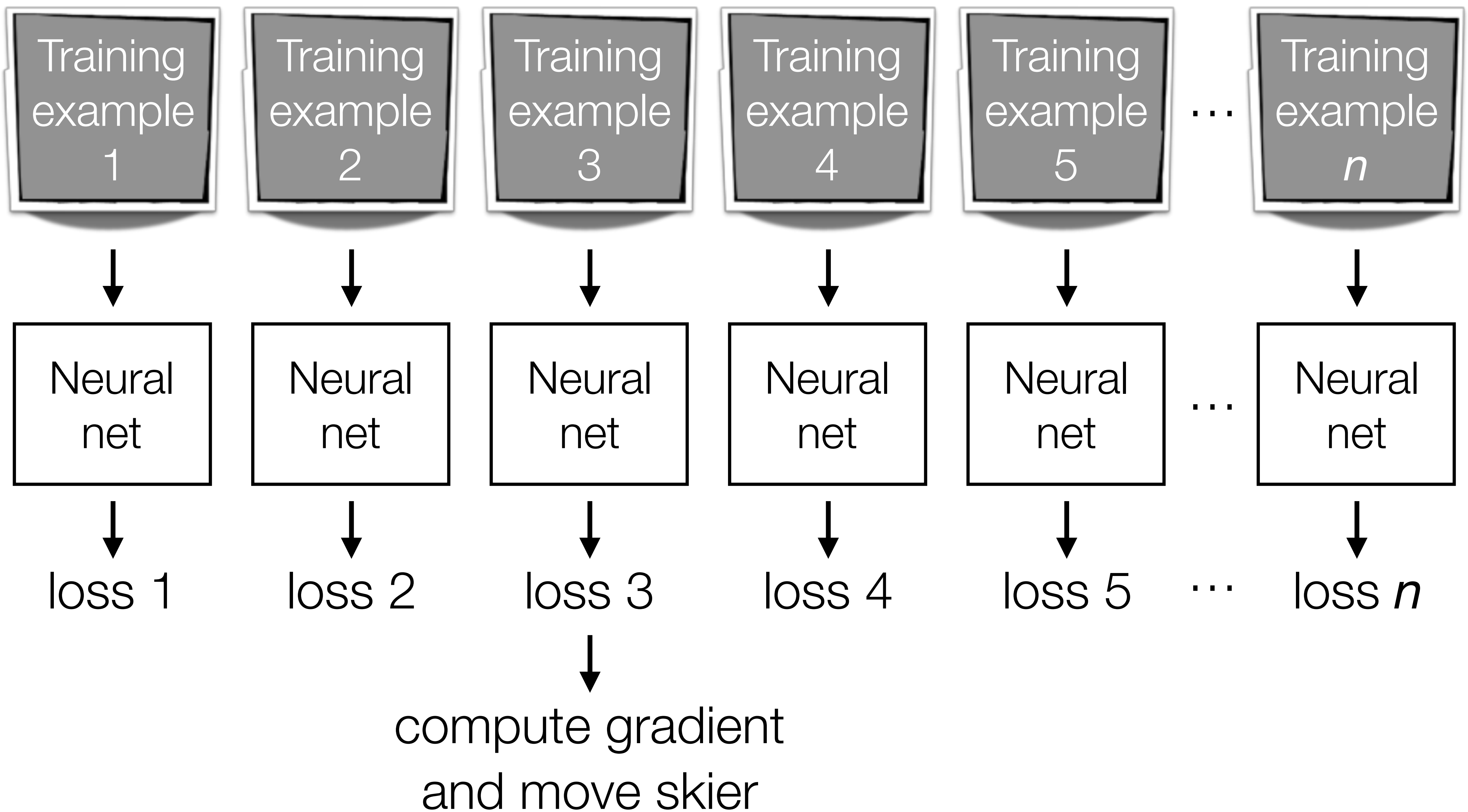

SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the “full” gradient)

# Stochastic Gradient Descent (SGD)



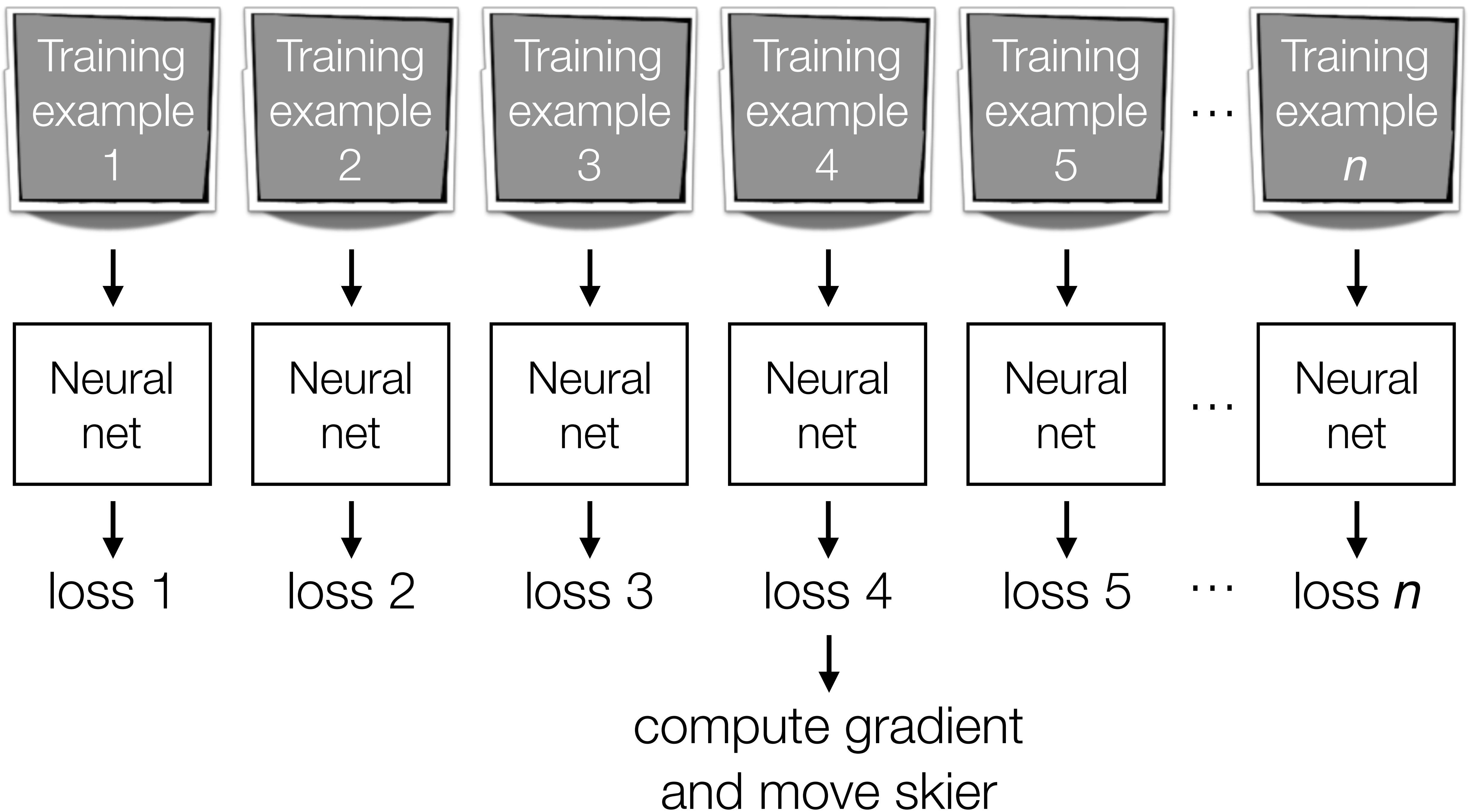

SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the “full” gradient)

# Stochastic Gradient Descent (SGD)



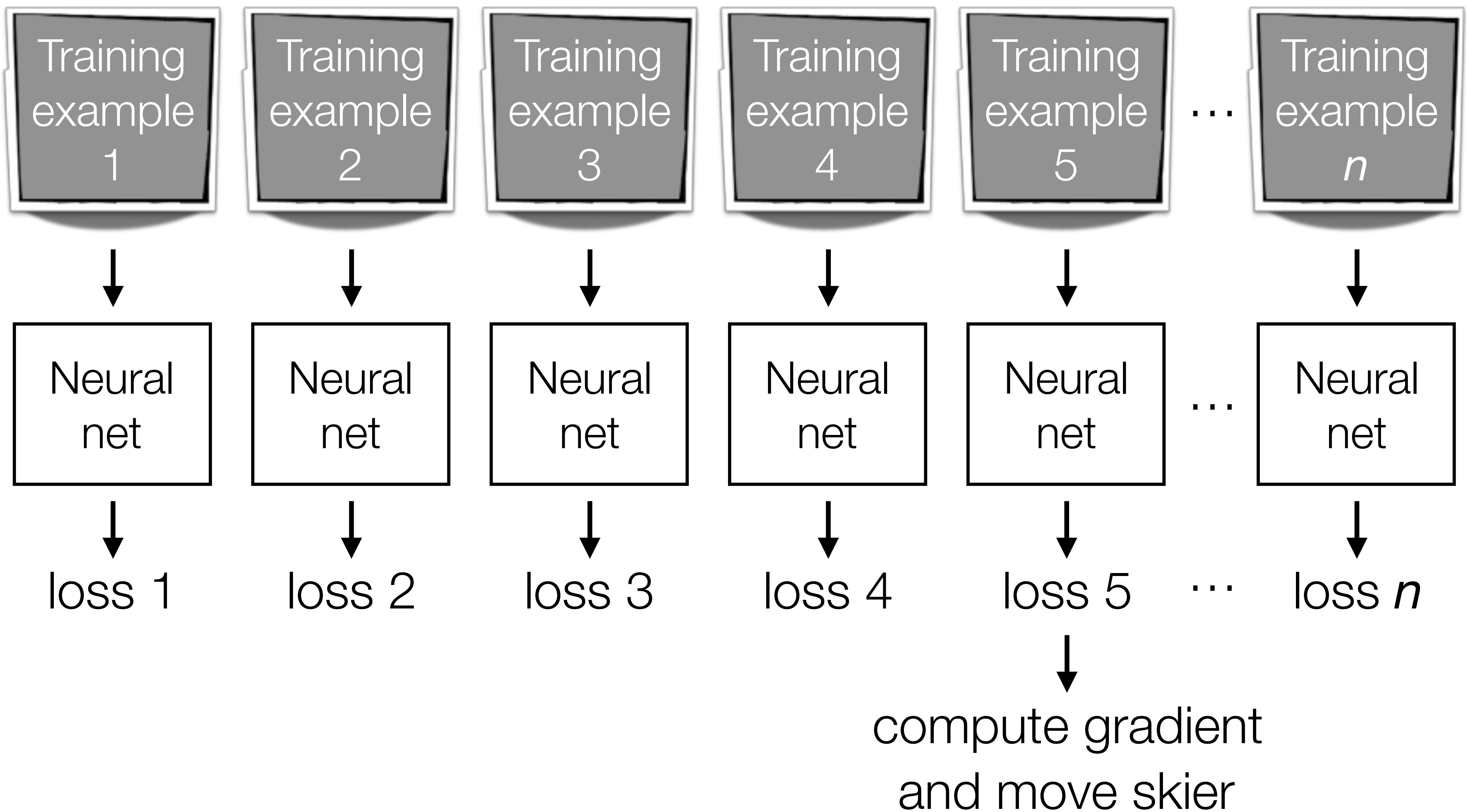

SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

# Stochastic Gradient Descent (SGD)



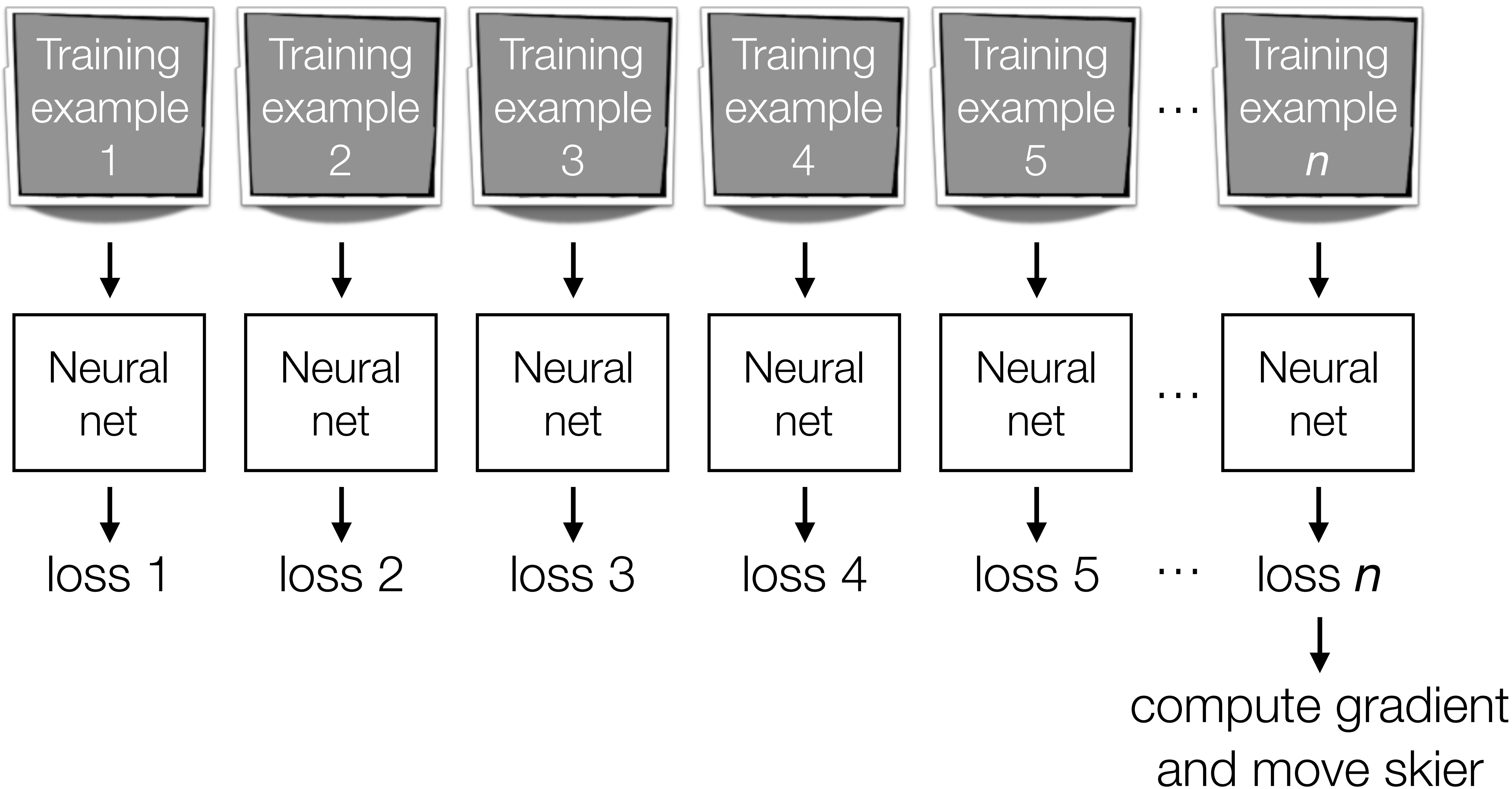

SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

# Stochastic Gradient Descent (SGD)



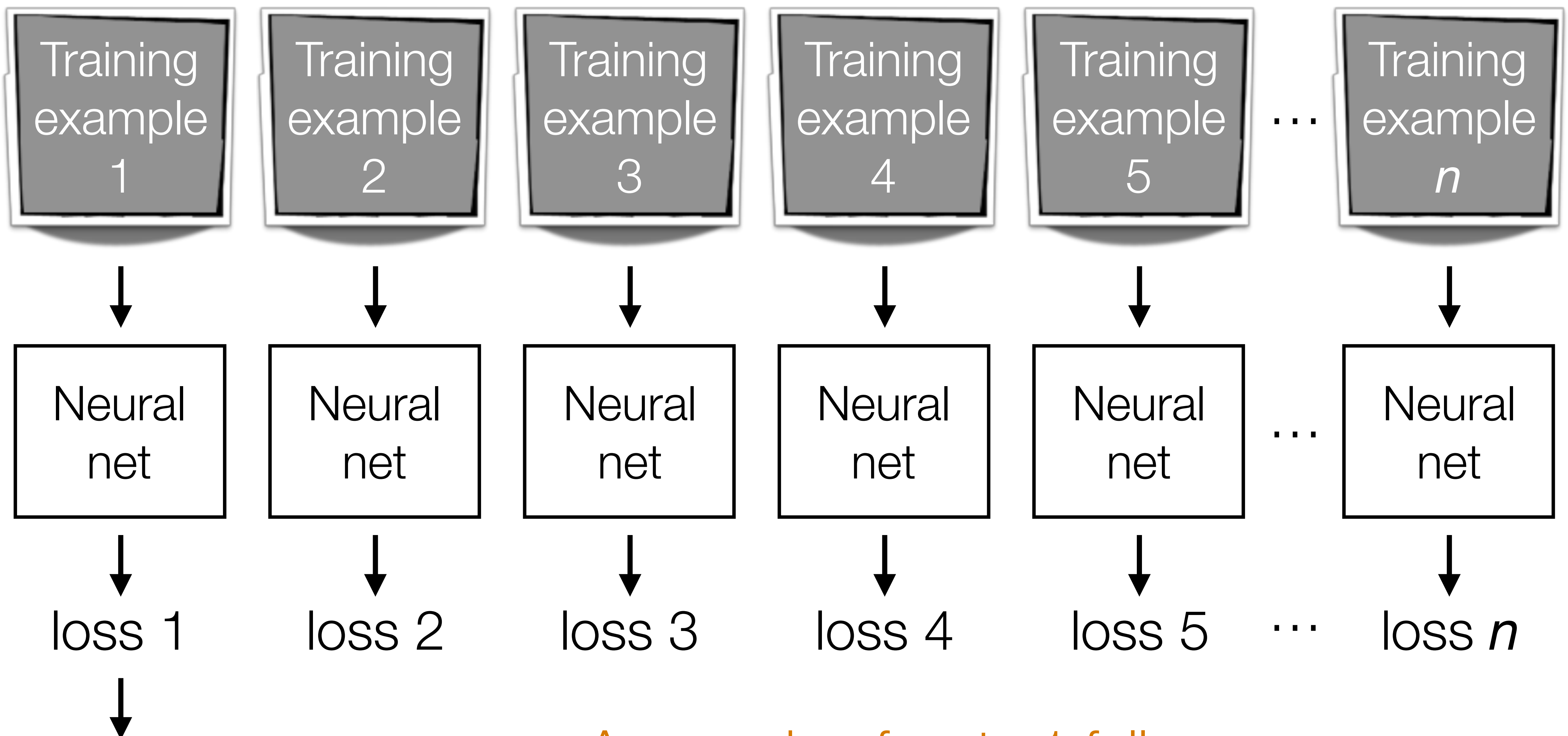

SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

# Stochastic Gradient Descent (SGD)




SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

# Stochastic Gradient Descent (SGD)

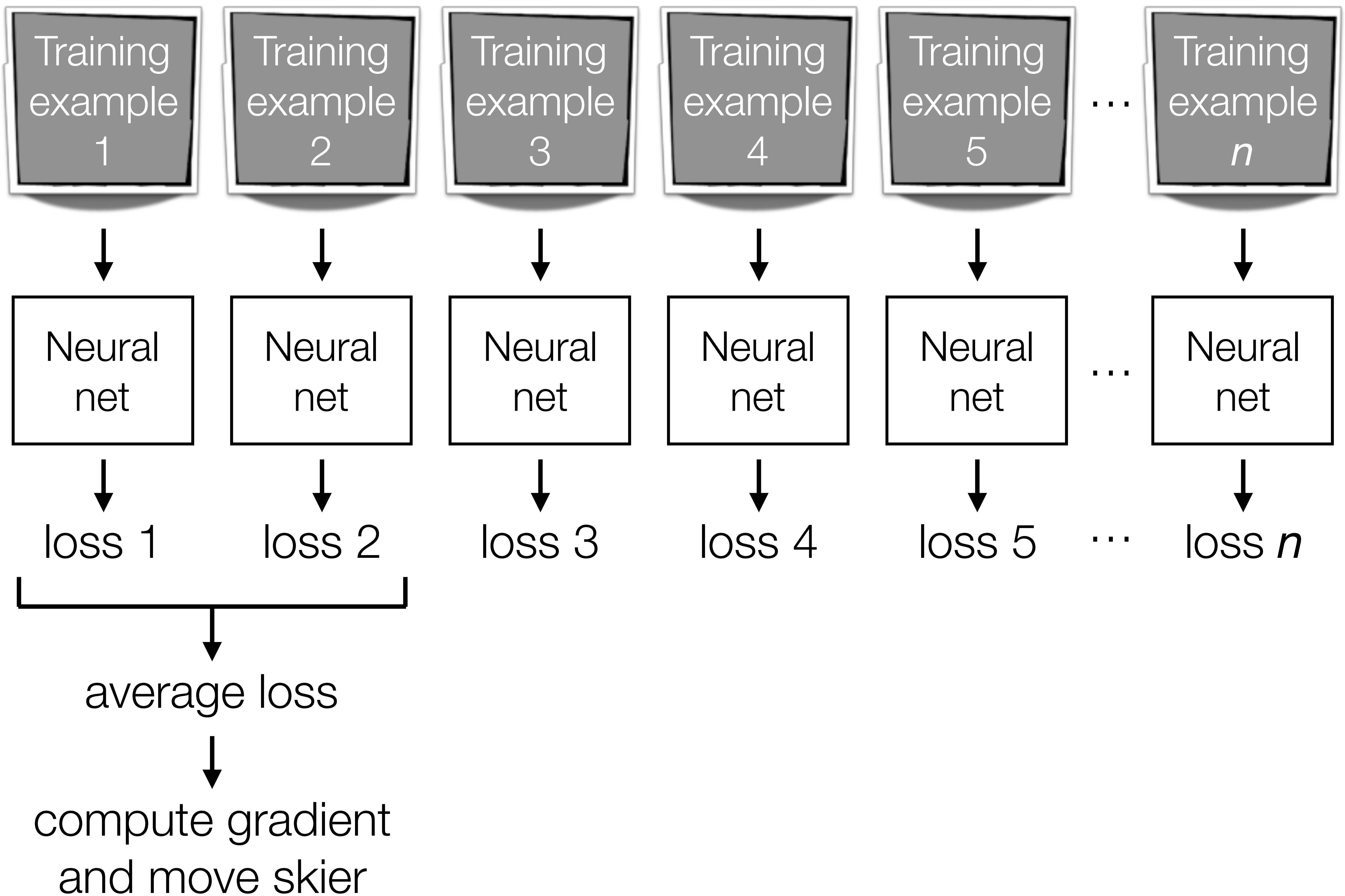



compute gradient
and move skier

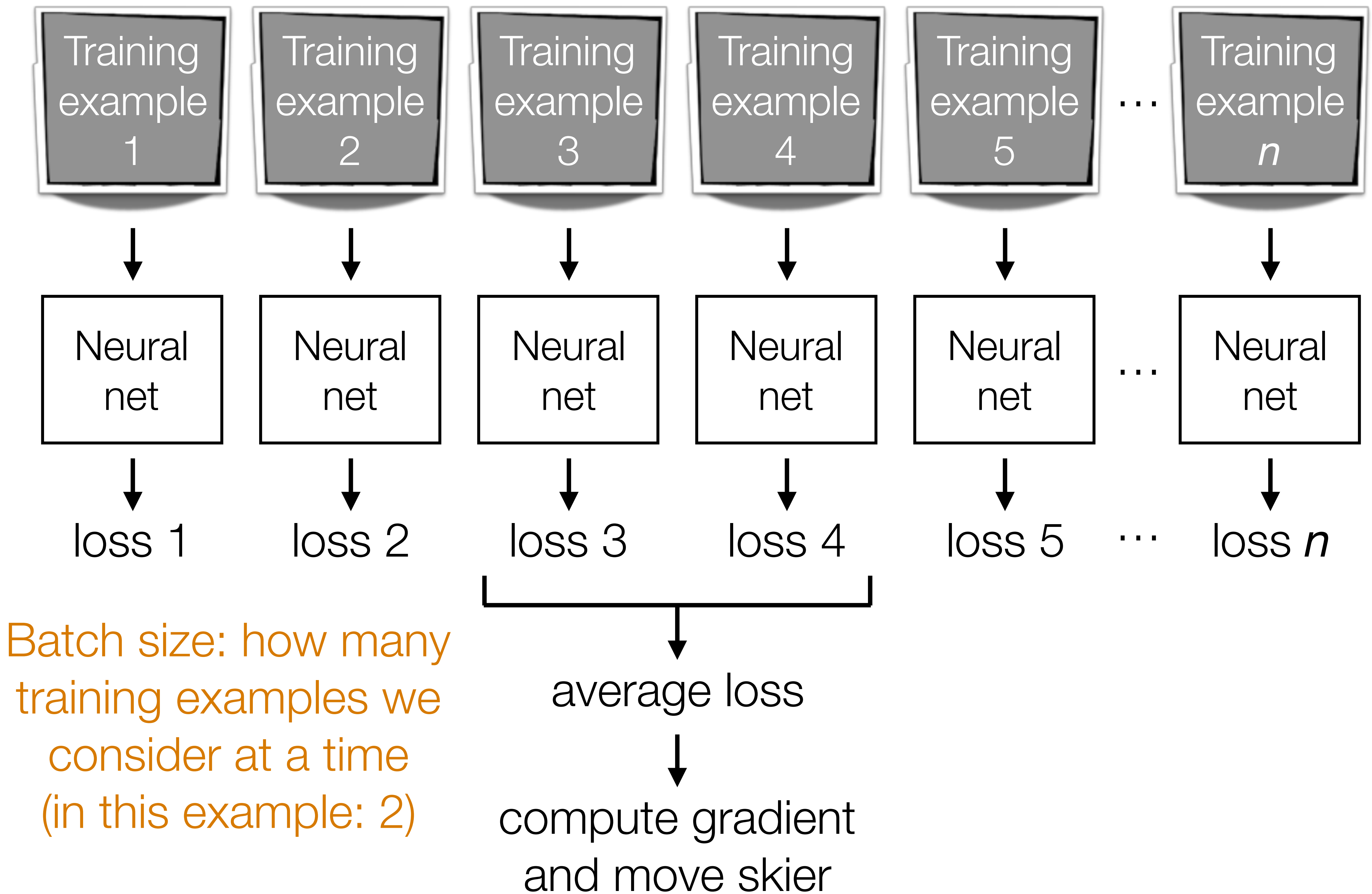
An epoch refers to 1 full pass through all the training data

SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

# Mini-Batch Gradient Descent



# Mini-Batch Gradient Descent



## Best variant of SGD to use?
## Best # of epochs? Best batch size?

Active area of research

Depends on problem, data, hardware, etc

Example: even with a GPU, you can get slow learning (slower than CPU!) if you choose # epochs/batch size poorly!!!

# Dealing with Small Datasets

**Data augmentation:** generate perturbed versions of your training data to get larger training dataset

Training image
Training label: cat

Mirrored
Still a cat!

Rotated & translated
Still a cat!

We just turned 1 training example in 3 training examples

Allowable perturbations depend on data (e.g., for handwritten digits, rotating by 180 degrees would be bad: confuse 6's and 9's)

# Dealing with Small Datasets

**Fine tuning:** if there's an existing pre-trained neural net, you could modify it for your problem that has a small dataset

**Example:** classify between Tesla's and Toyota's

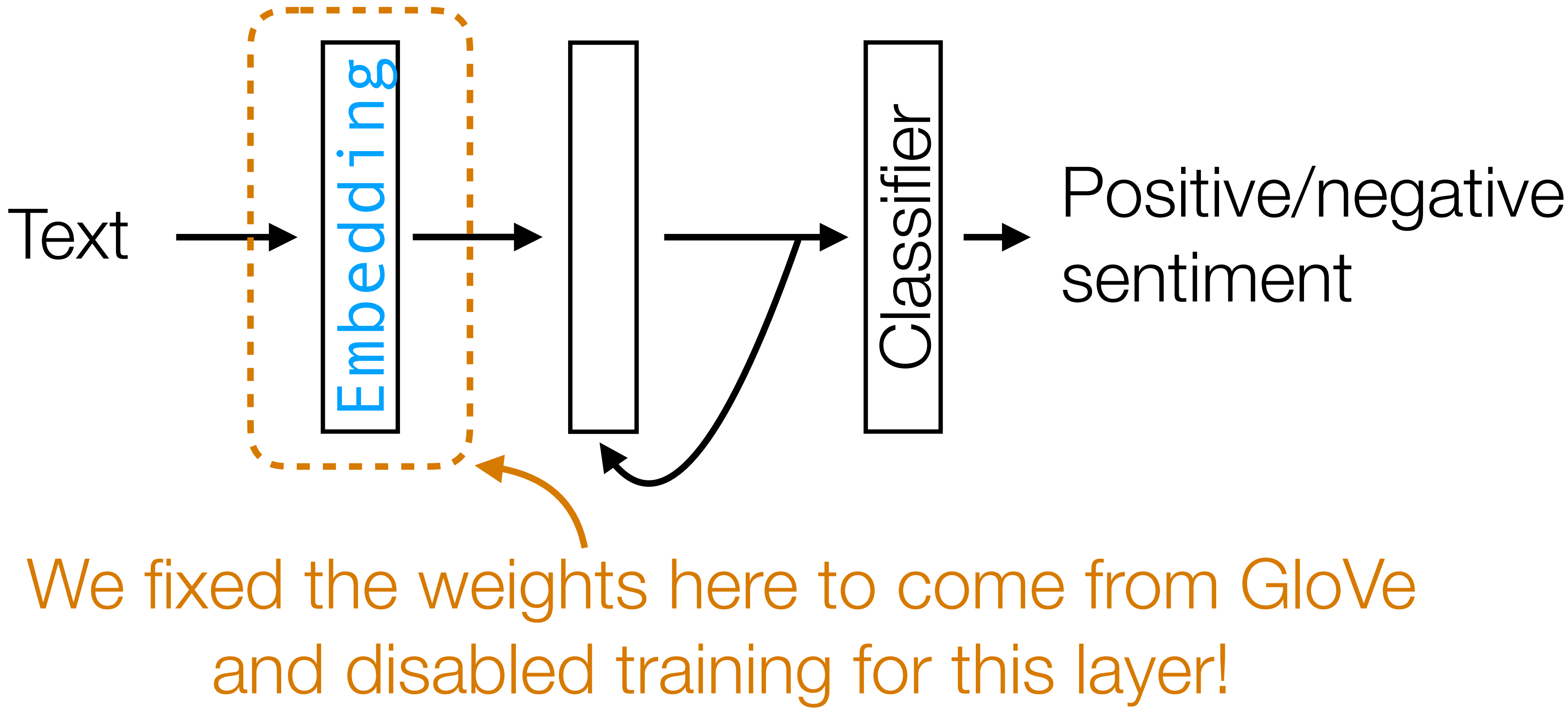
You collect photos from the internet of both, but your dataset size is small, on the order of 1000 images

Strategy: take existing pre-trained CNN for ImageNet classification and change final layer to do classification between Tesla's and Toyota's rather than classifying into 1000 objects

# Dealing with Small Datasets

**Fine tuning:** if there's an existing pre-trained neural net, you could modify it for your problem that has a small dataset

**Example:** sentiment analysis RNN demo



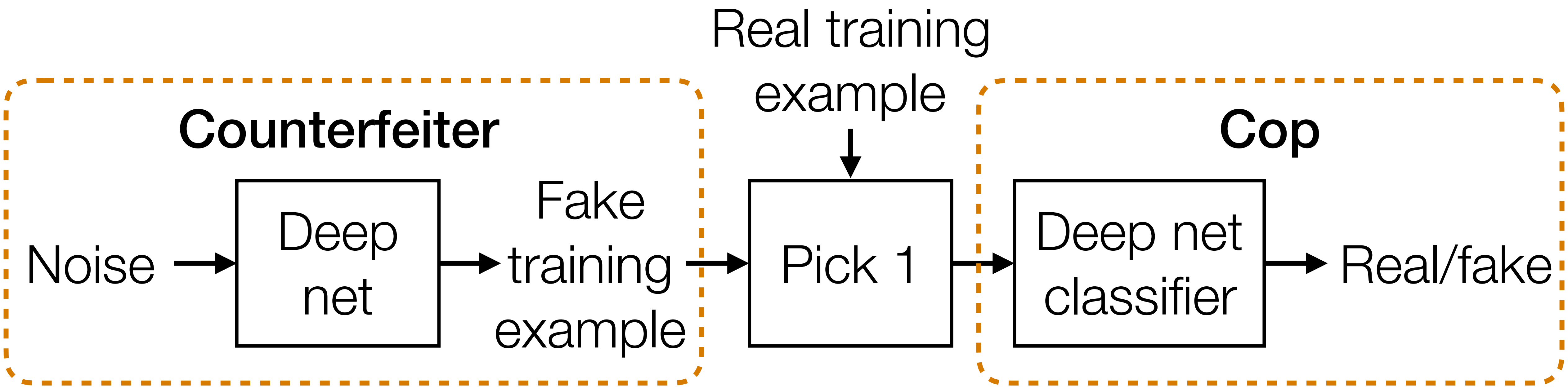

GloVe vectors pre-trained on massive dataset (Wikipedia + Gigaword)

IMDb review dataset is small in comparison

# Generate Fake Data that Look Real

Unsupervised approach: generate data that look like training data

**Example:** Generative Adversarial Network (GAN)




Counterfeiter tries to get better at tricking the cop

Cop tries to get better at telling which examples are real vs fake

Terminology: counterfeiter is the **generator**, cop is the **discriminator**

Other approaches: variational autoencoders, pixelRNNs/pixelCNNs

# Generate Fake Data that Look Real

Fake celebrities generated by NVIDIA using GANs
(Karras et al Oct 27, 2017)

Google DeepMind's WaveNet makes fake audio that sounds like whoever you want using pixelRNNs (Oord et al 2016)

# Generate Fake Data that Look Real

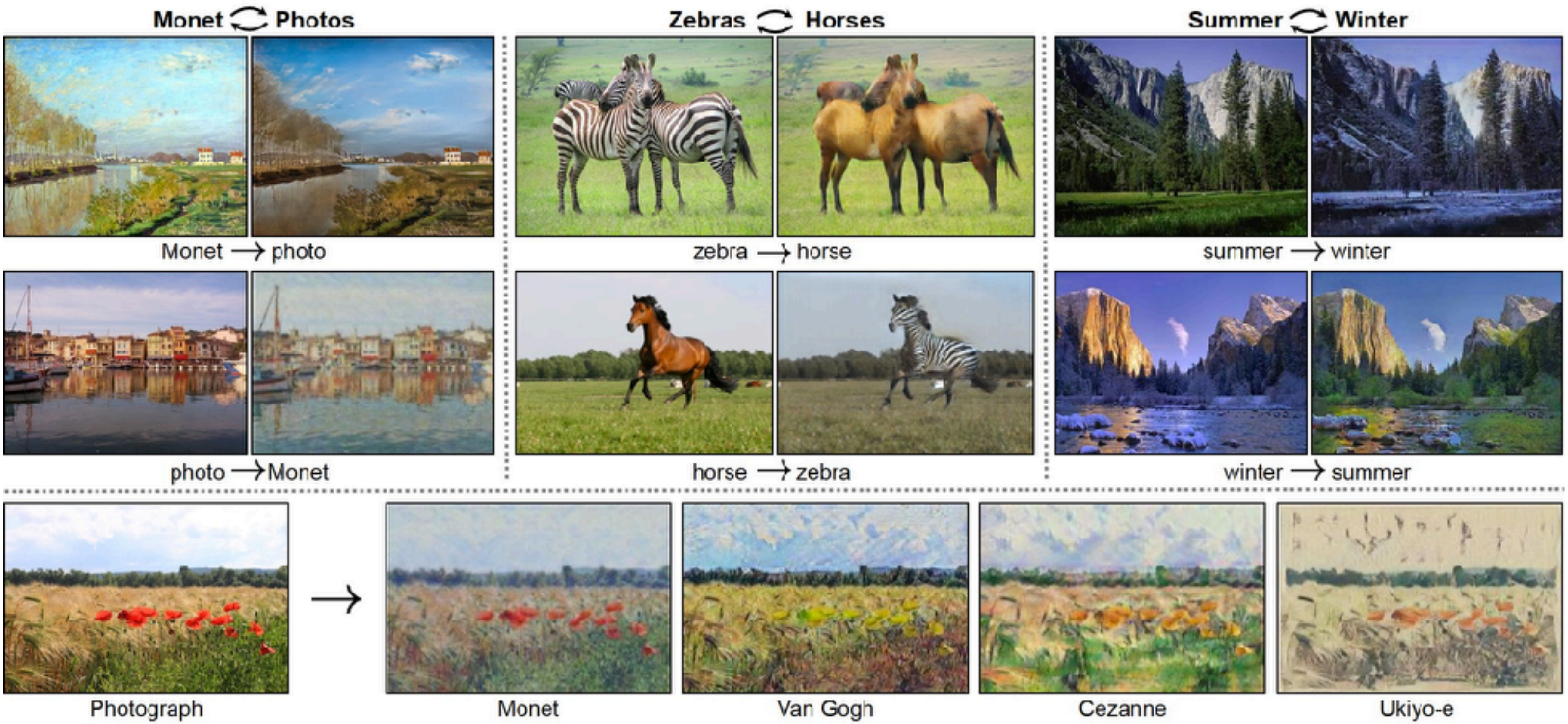
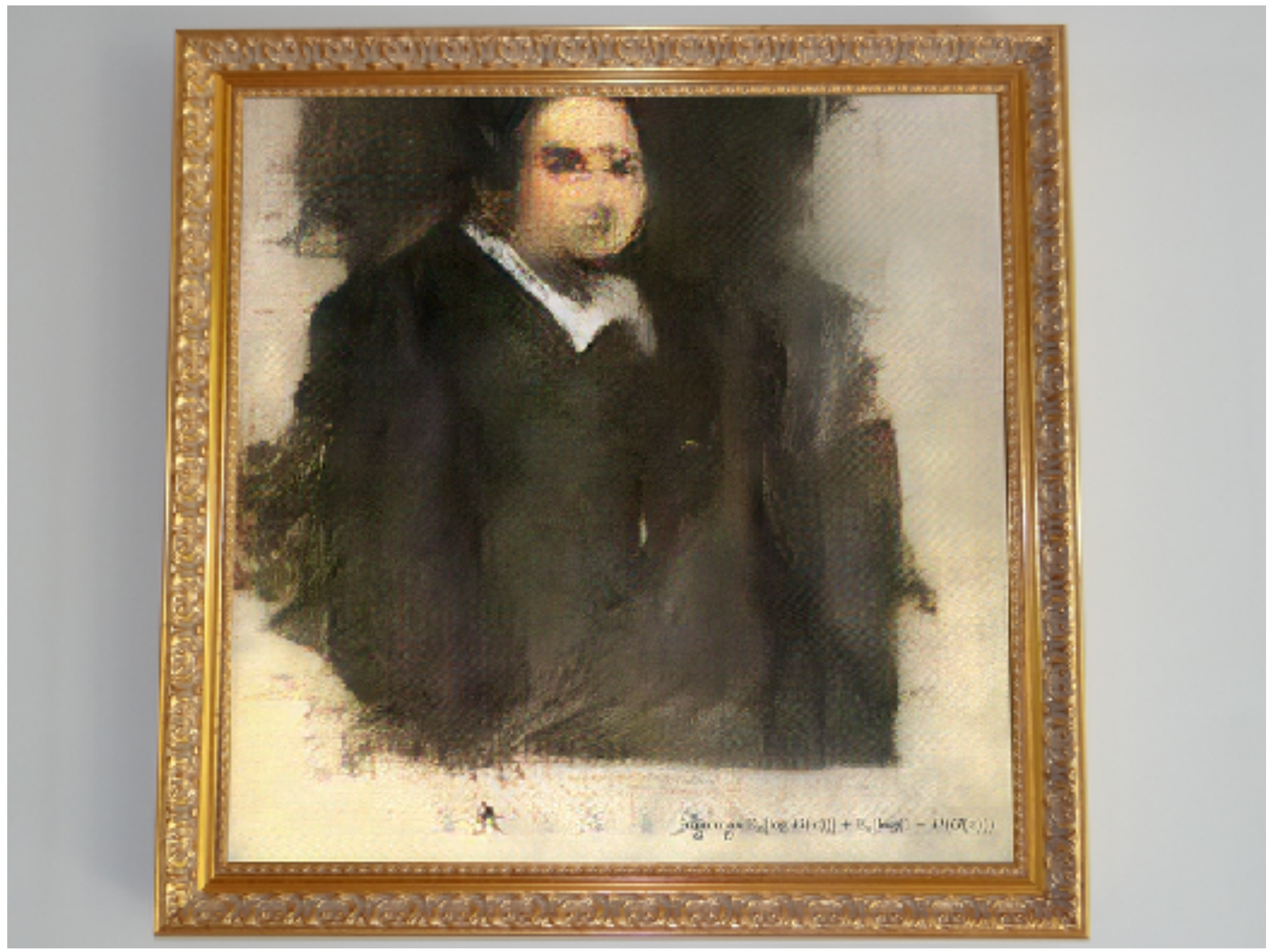



Image-to-image translation results from UC Berkeley using GANs
(Isola et al 2017, Zhu et al 2017)

# Generate Fake Art



October 2018: estimated to go for $7,000-$10,000

**10/25/2018: Sold for $432,500**

Source: https://www.npr.org/2018/10/22/659680894/a-i-produced-portrait-will-go-up-for-auction-at-christie-s

# AI News Anchor

## China's Xinhua agency unveils AI news presenter

By Chris Baraniuk
Technology reporter

8 November 2018

Share


Source: https://www.bbc.com/news/technology-46136504

# Harrison Ford as Young Han Solo

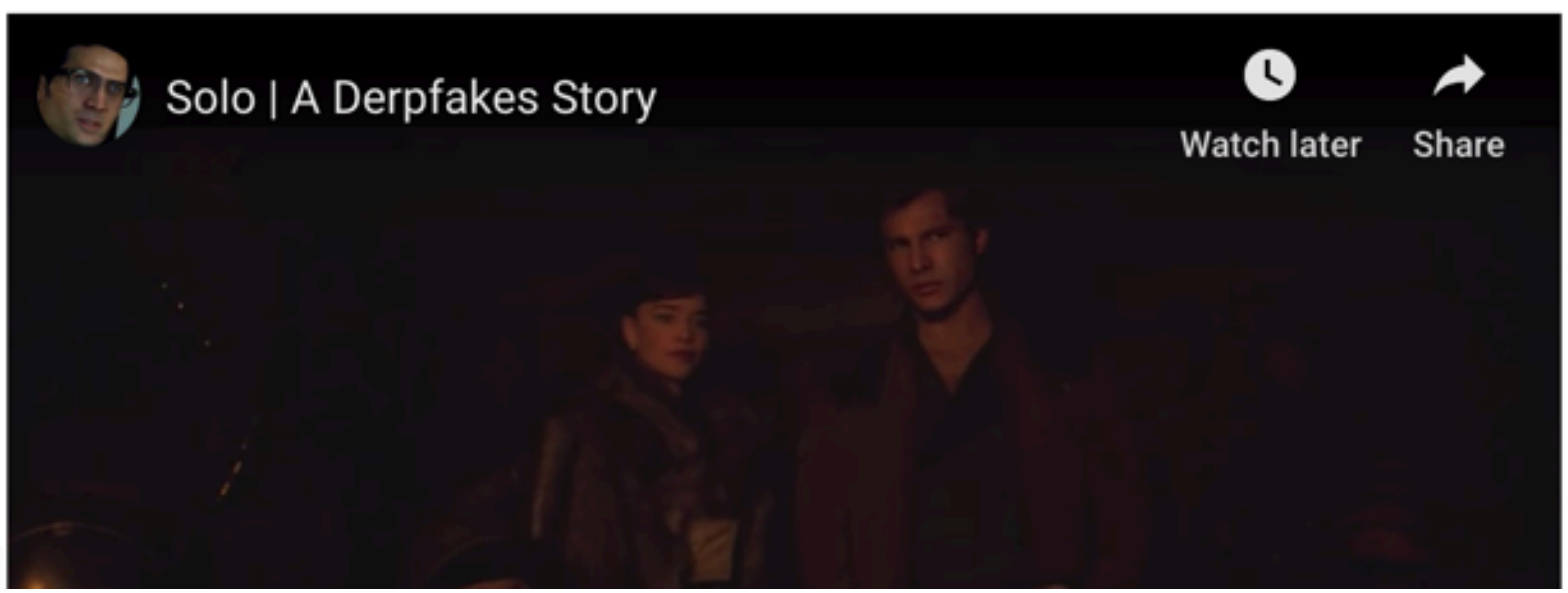
10

## Deepfake edits have put Harrison Ford into Solo: A Star Wars Story, for better or for worse

*Uncanny valley, here we come*

By Chaim Gartenberg | @cgartenberg | Oct 17, 2018, 3:37pm EDT

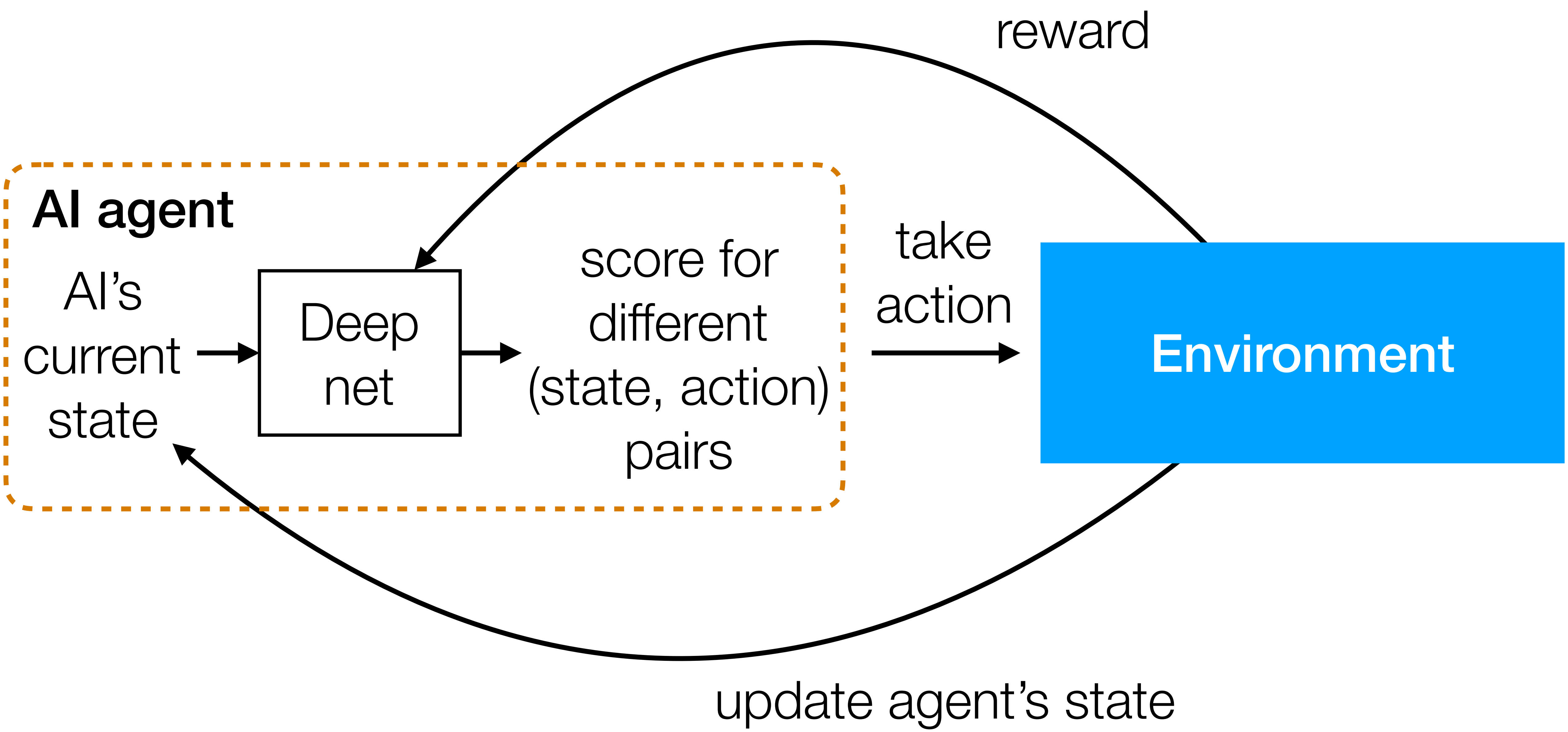
SHARE




Source: https://www.theverge.com/2018/10/17/17990162/deepfake-edits-harrison-ford-han-solo-a-star-wars-story-alden-ehrenreich

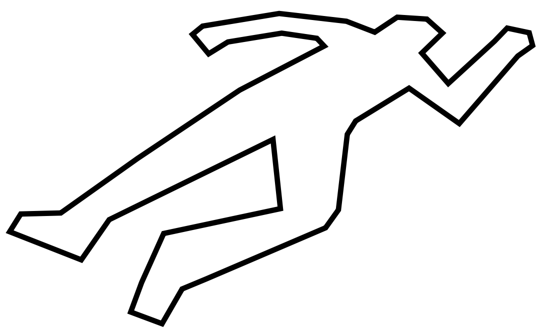
# Deep Reinforcement Learning

The machinery behind AlphaGo and similar systems



# Unstructured Data Analysis

Question



*The dead body*

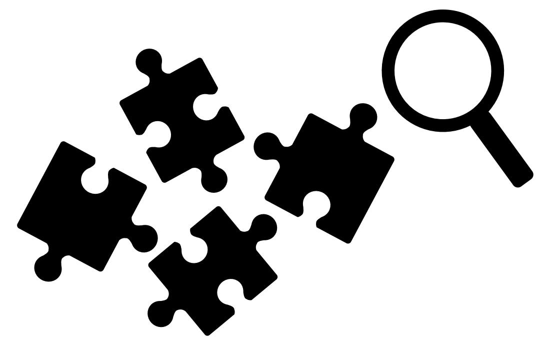
This is provided by a practitioner

Data

*The evidence*

Some times you have to collect more evidence!

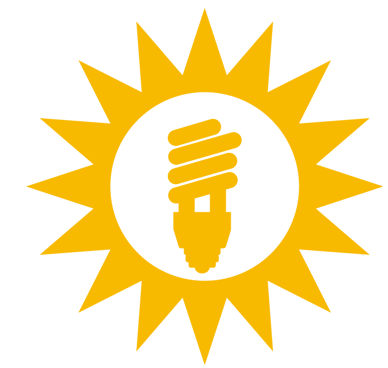
Finding Structure



*Puzzle solving, careful analysis*

Exploratory data analysis

Insights



*When? Where? Why? How? Perpetrator catchable?*

Answer original question

<u>There isn't always a follow-up prediction problem to solve</u>

# Some Parting Thoughts

- Remember to **visualize steps of your data analysis pipeline**
  - Helpful in debugging & interpreting intermediate/final outputs
- Very often there are *tons* of models/design choices to try
  - Come up with **quantitative metrics** that make sense for your problem, and use these metrics to **evaluate models (think about how we chose hyperparameters!)**
  - But don’t blindly rely on metrics without **interpreting results in the context of your original problem!**
- Often times you won’t have labels! If you really want labels:
  - Manually obtain labels (either you do it or crowdsource)
  - Set up “self-supervised” learning task
- There is a *lot* we did not cover — **keep learning!**