Carnegie Mellon University
Heinz College

95-865 Unstructured Data Analytics

Recitation: Gradient descent, more on RNNs and time series analysis

Slides by George H. Chen

Learning a Deep Net

Learning a Deep Net

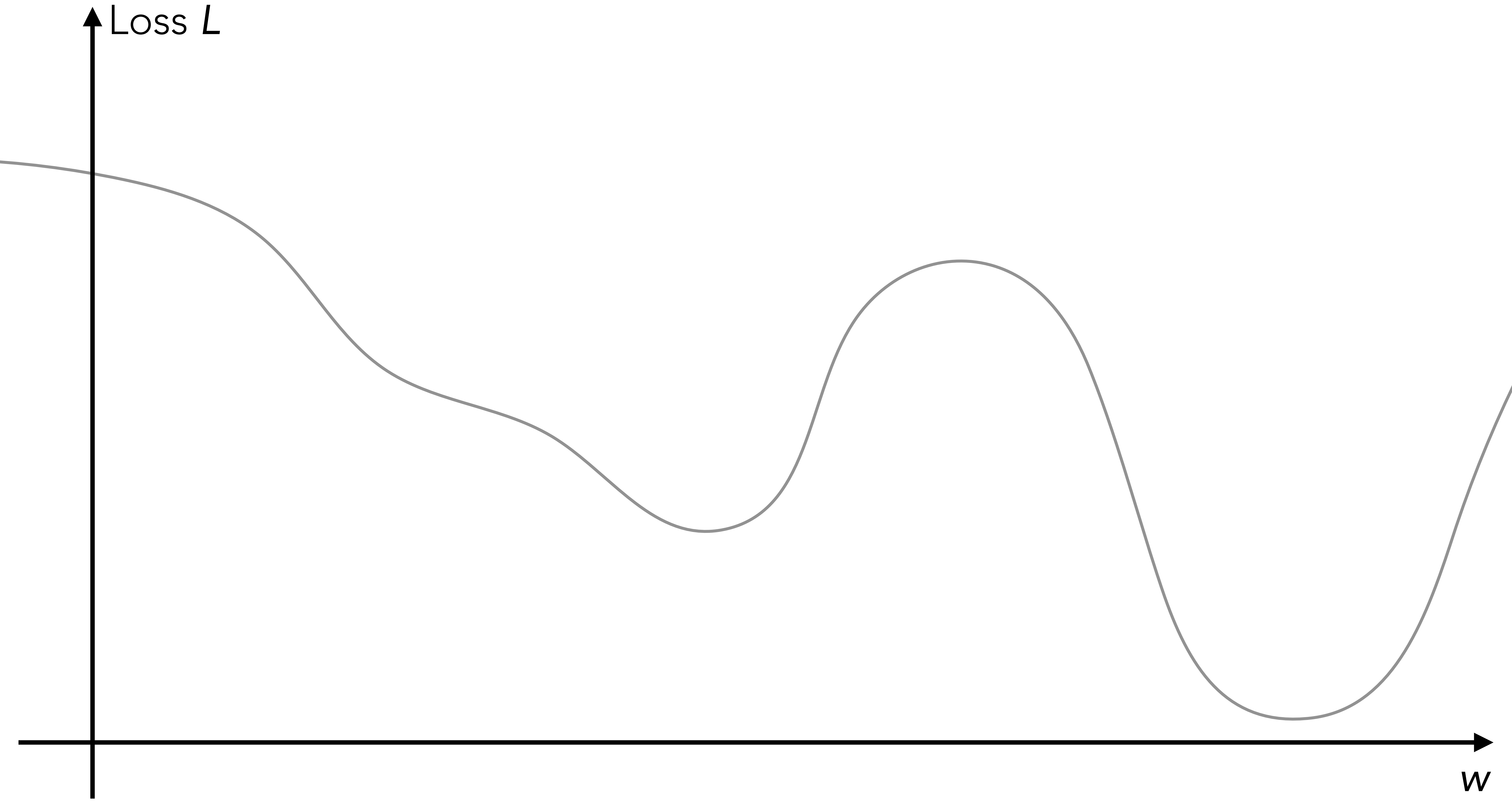
Suppose the neural network has a single real number parameter w

Learning a Deep Net

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

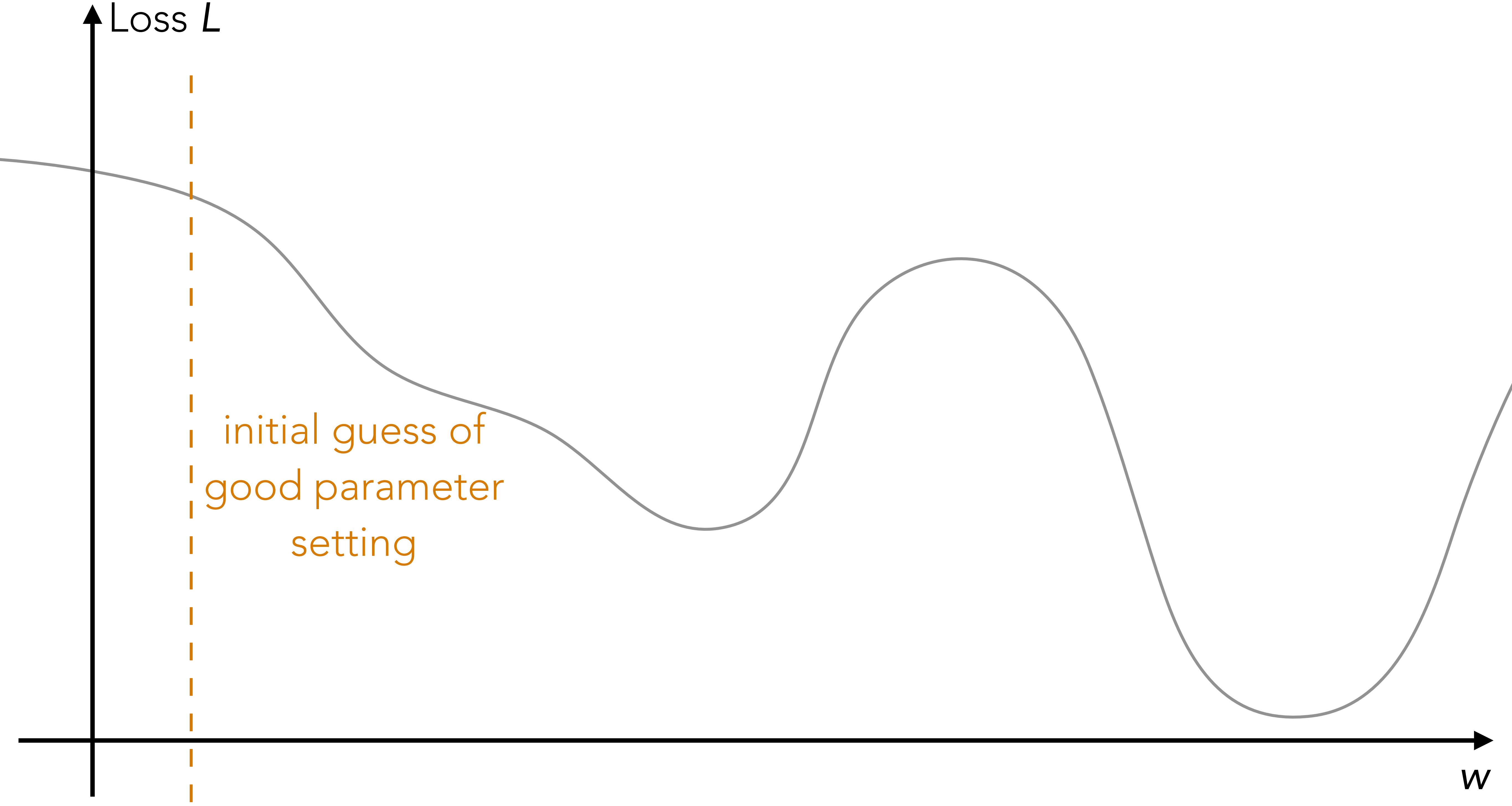
Learning a Deep Net

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



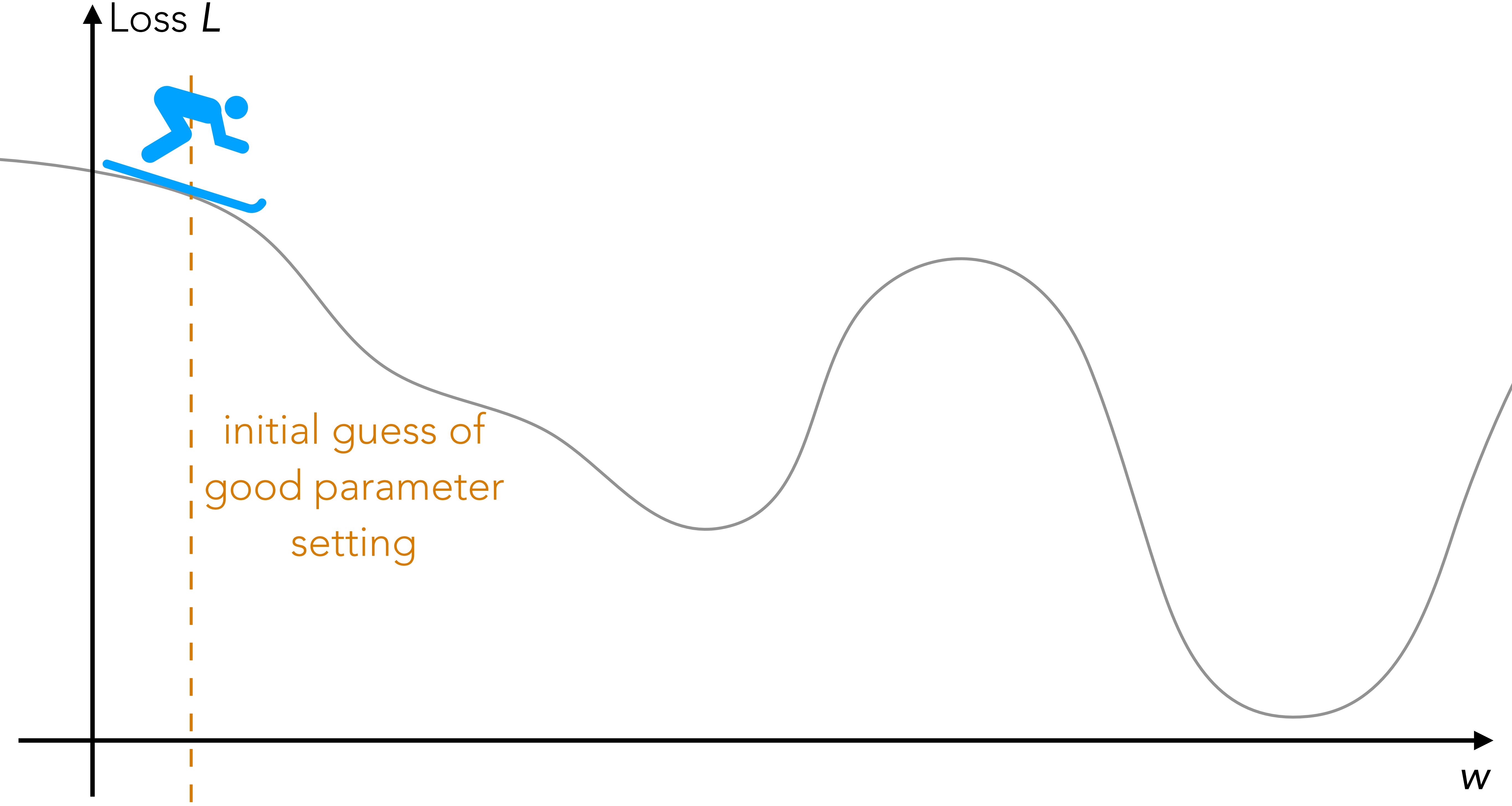
Learning a Deep Net

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



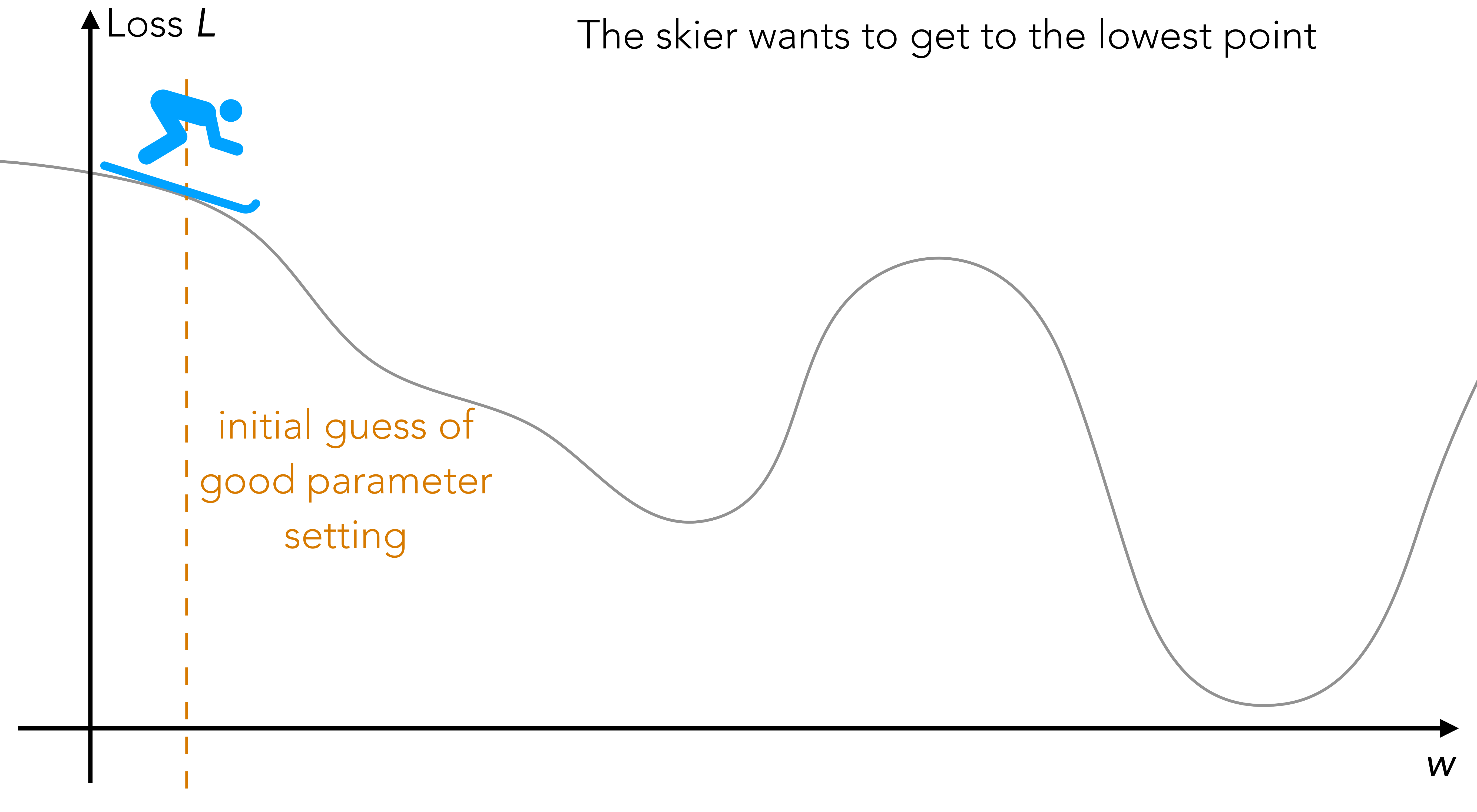
Learning a Deep Net

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***

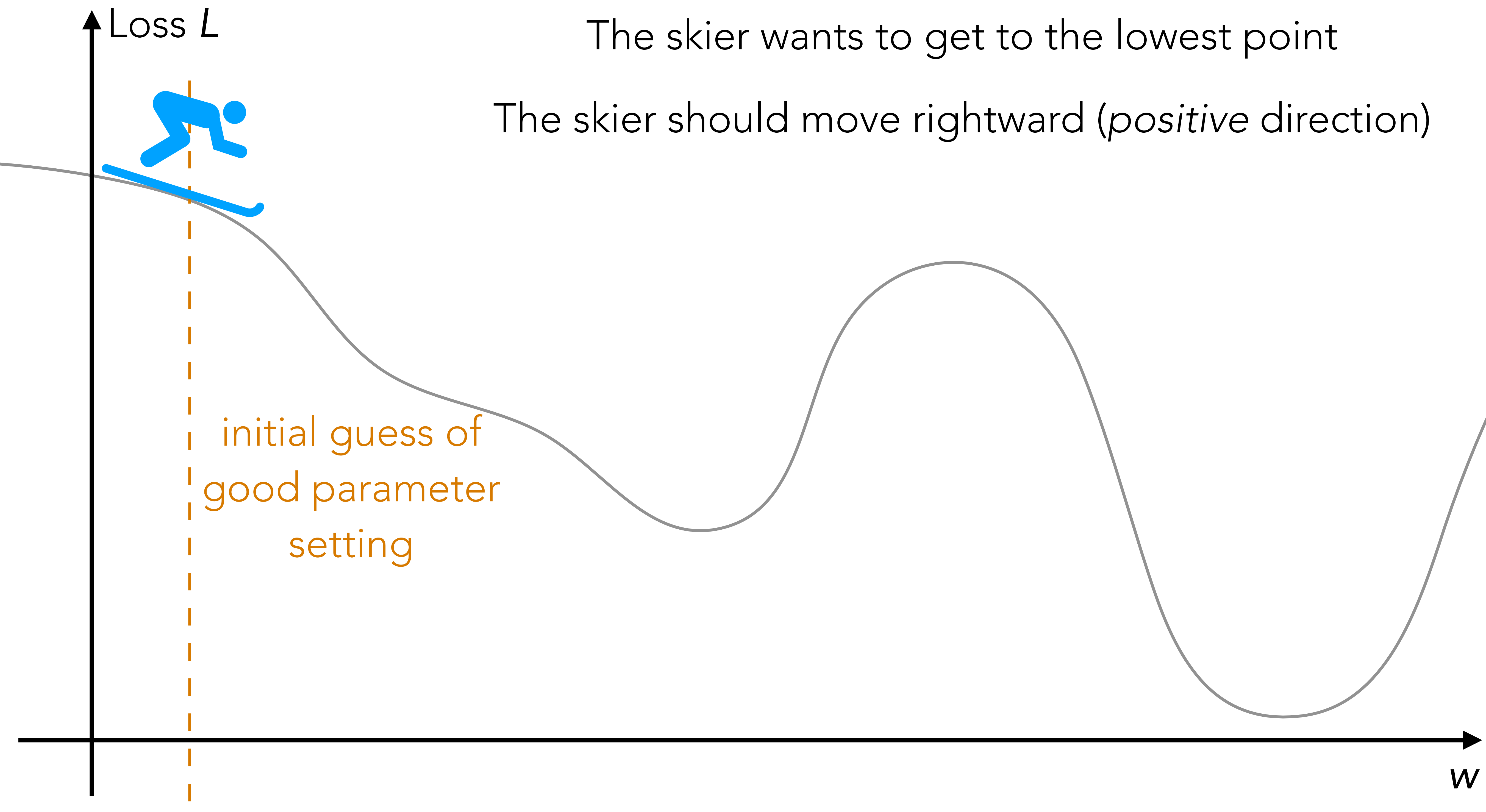


Learning a Deep Net

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

The skier wants to get to the lowest point

The skier should move rightward (*positive* direction)

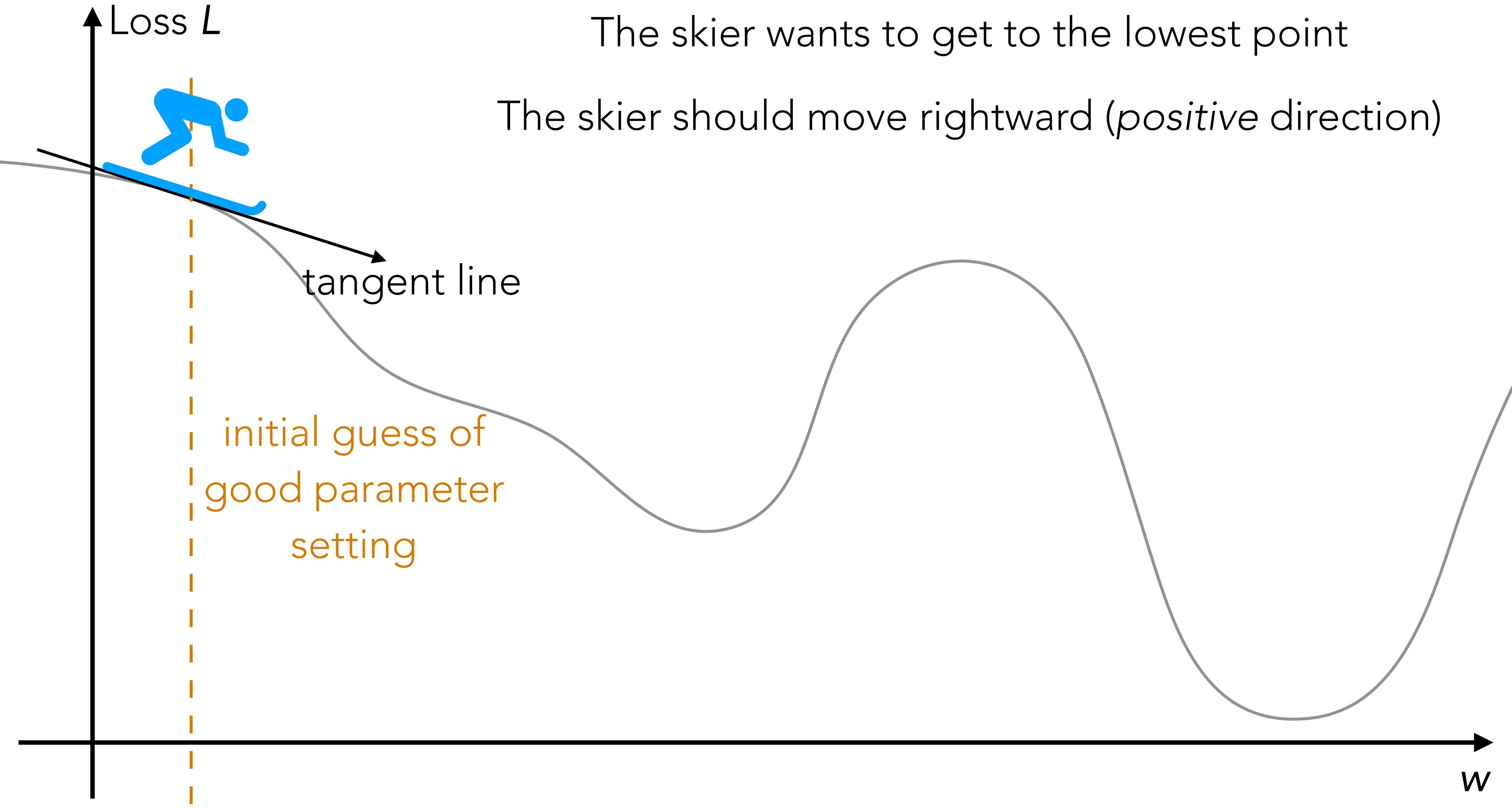


Learning a Deep Net

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

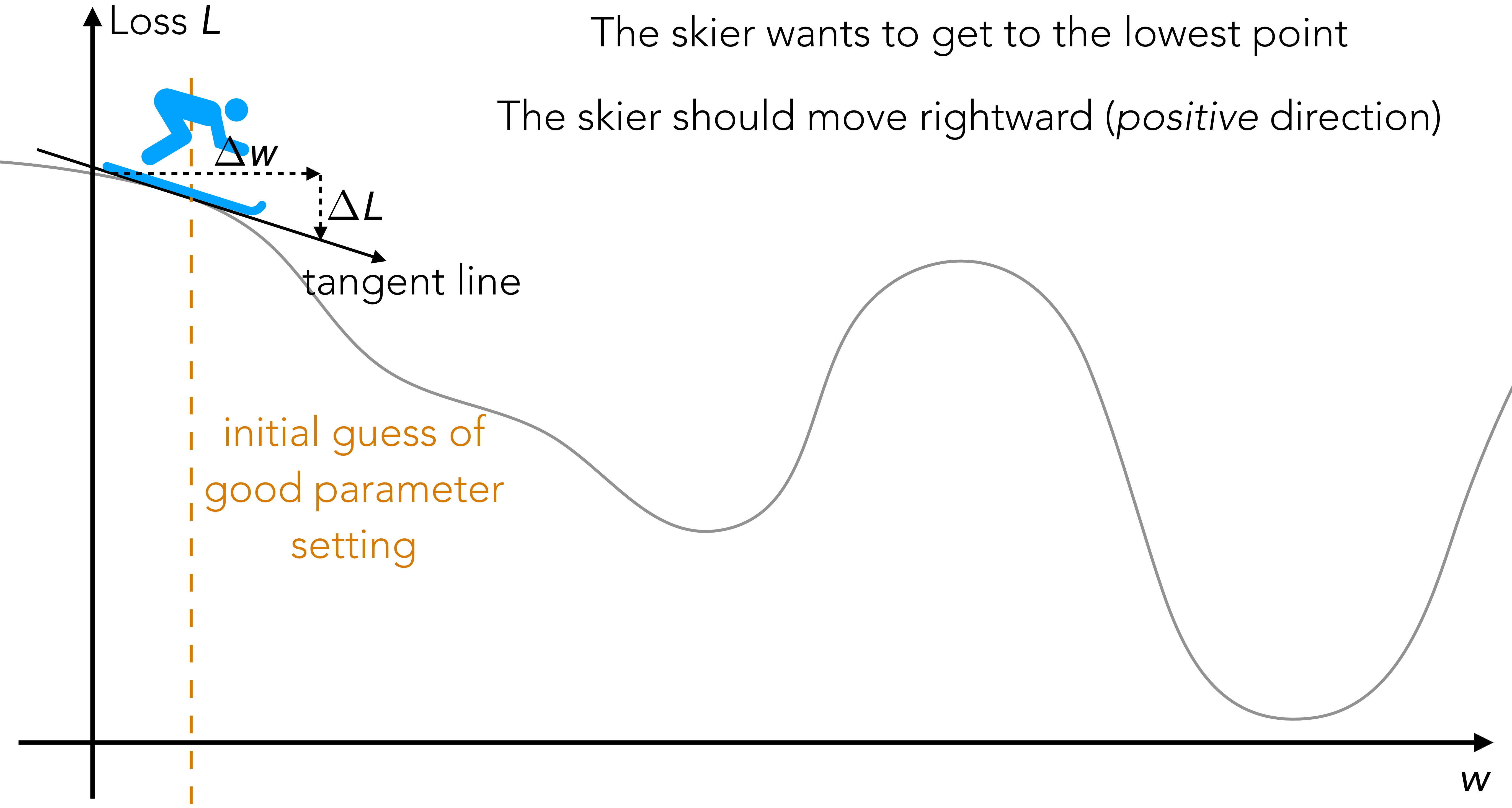
The skier wants to get to the lowest point

The skier should move rightward (*positive* direction)



Learning a Deep Net

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



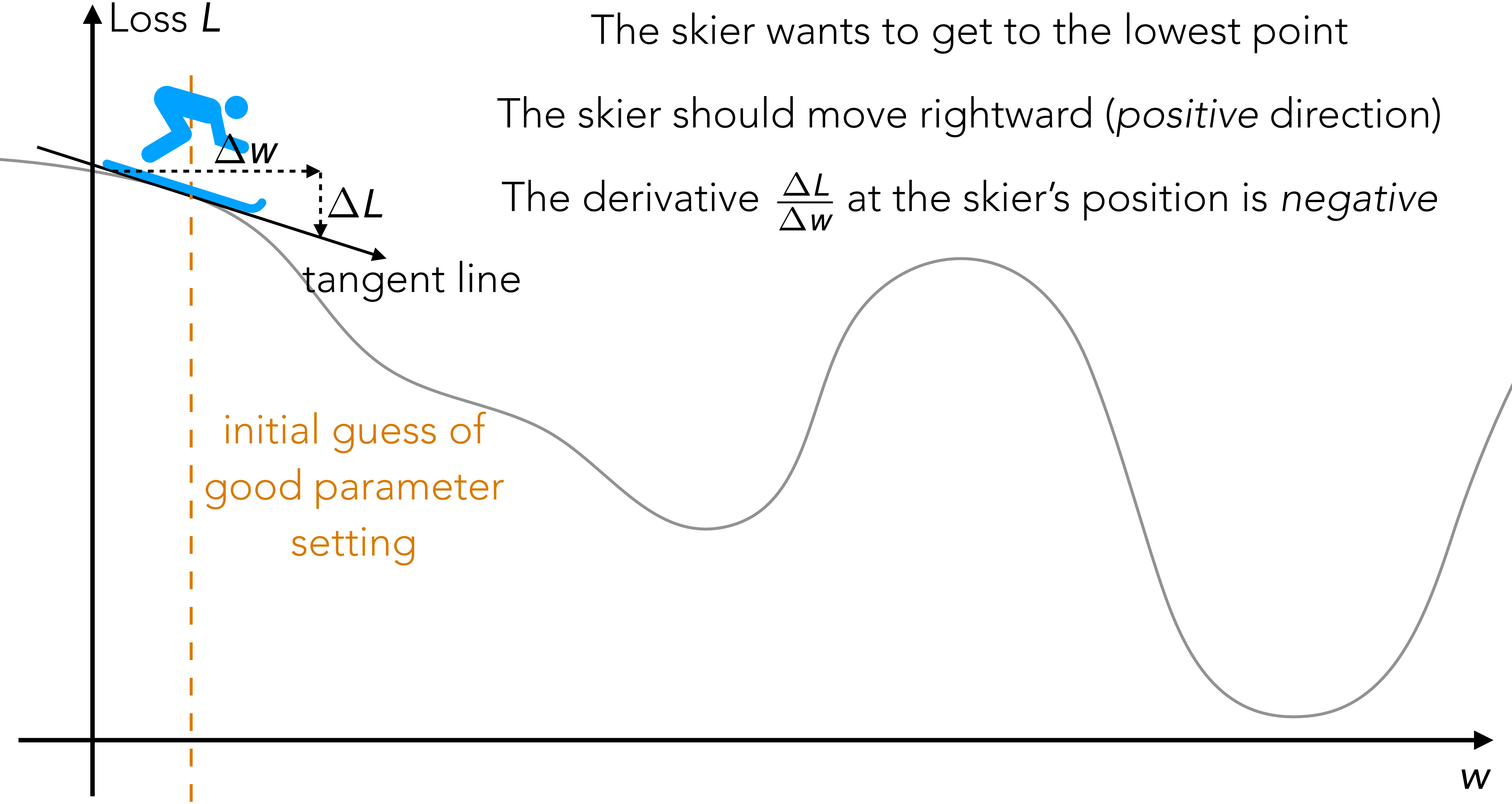
Learning a Deep Net

Suppose the neural network has a single real number parameter w

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*



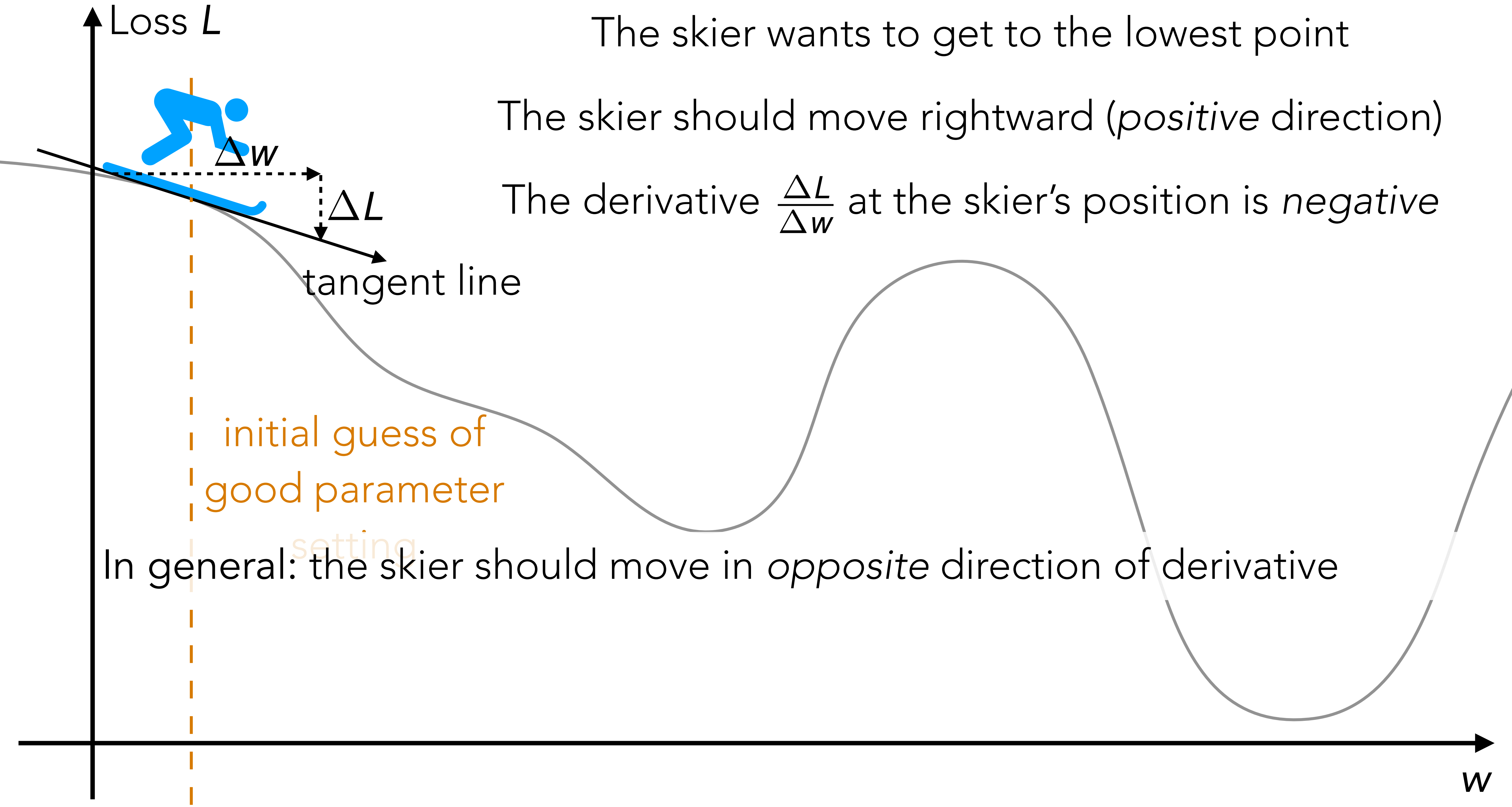
Learning a Deep Net

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)

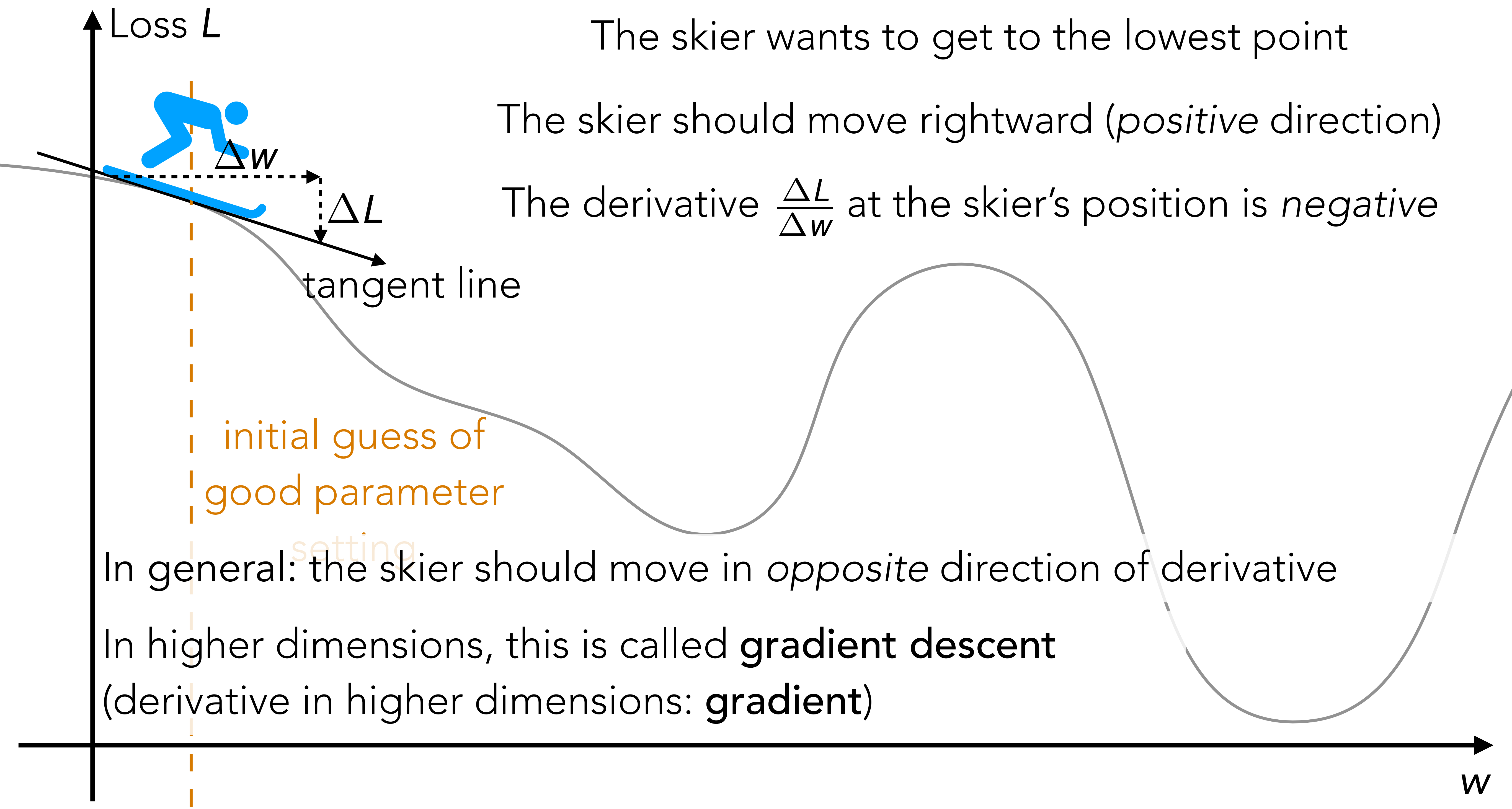
The derivative $\frac{\Delta L}{\Delta w}$ at the skier's position is *negative*



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

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)

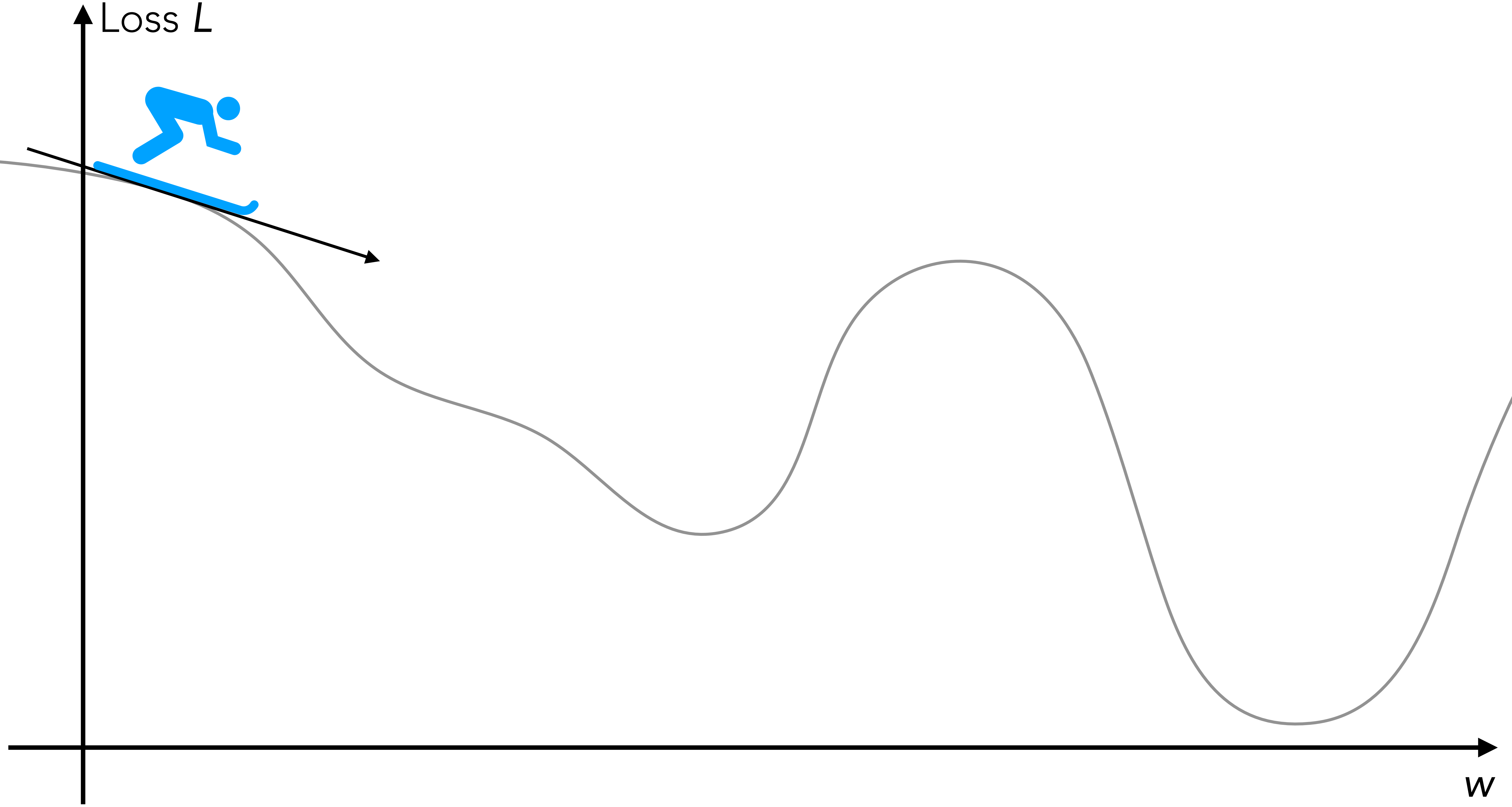
The derivative $\frac{\Delta L}{\Delta w}$ at the skier's position is *negative*

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

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

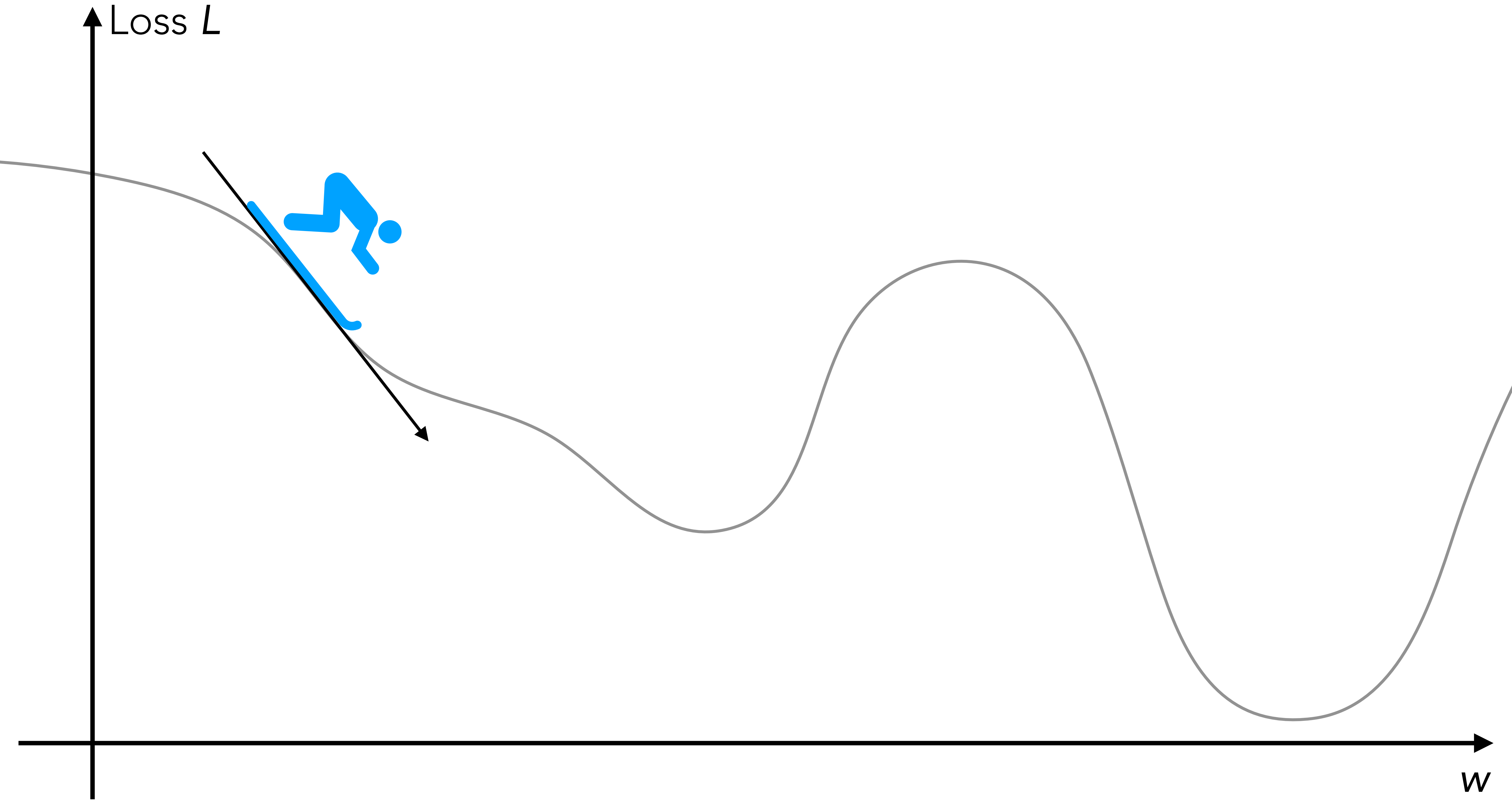
Learning a Deep Net

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



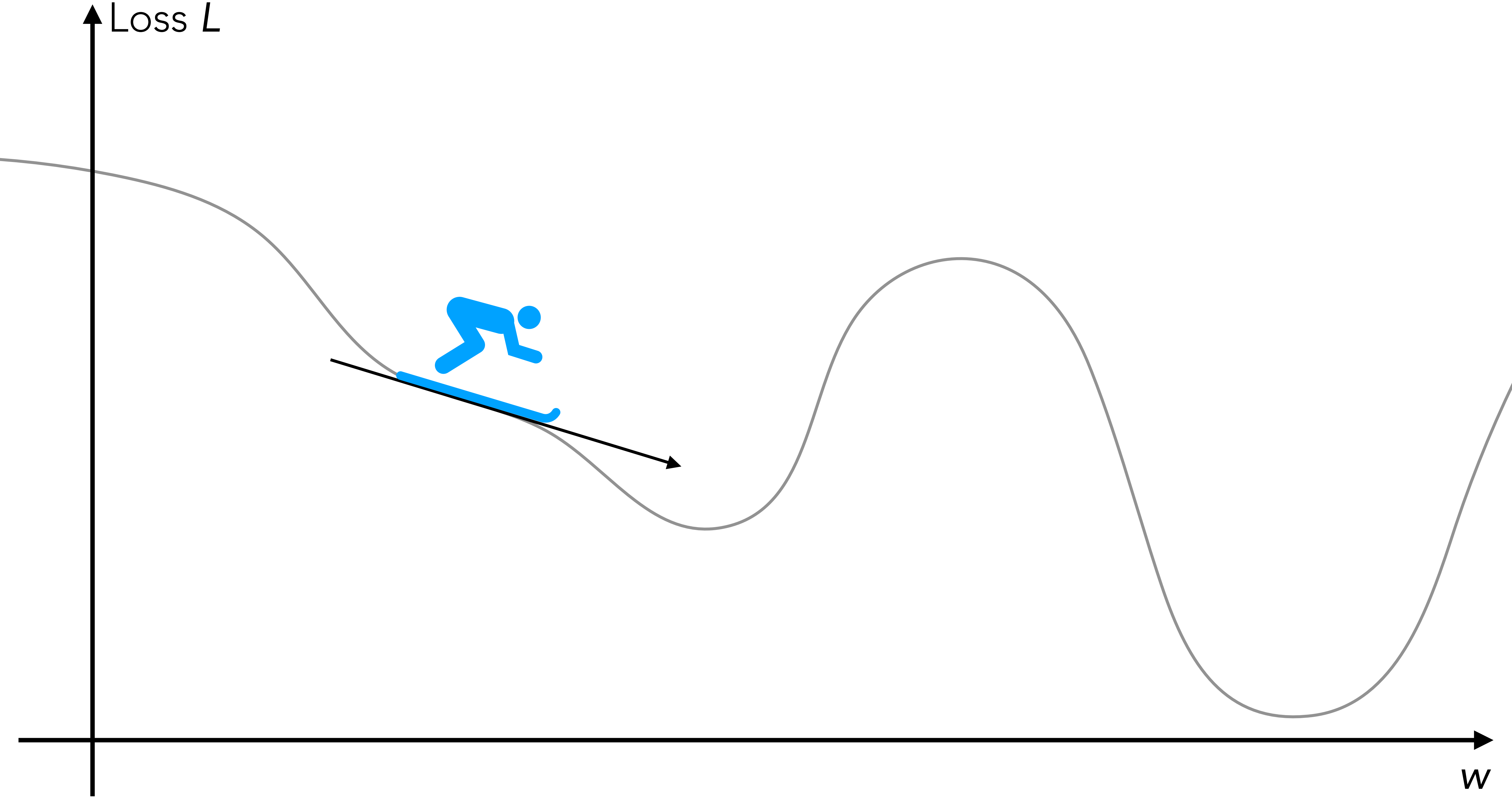
Learning a Deep Net

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



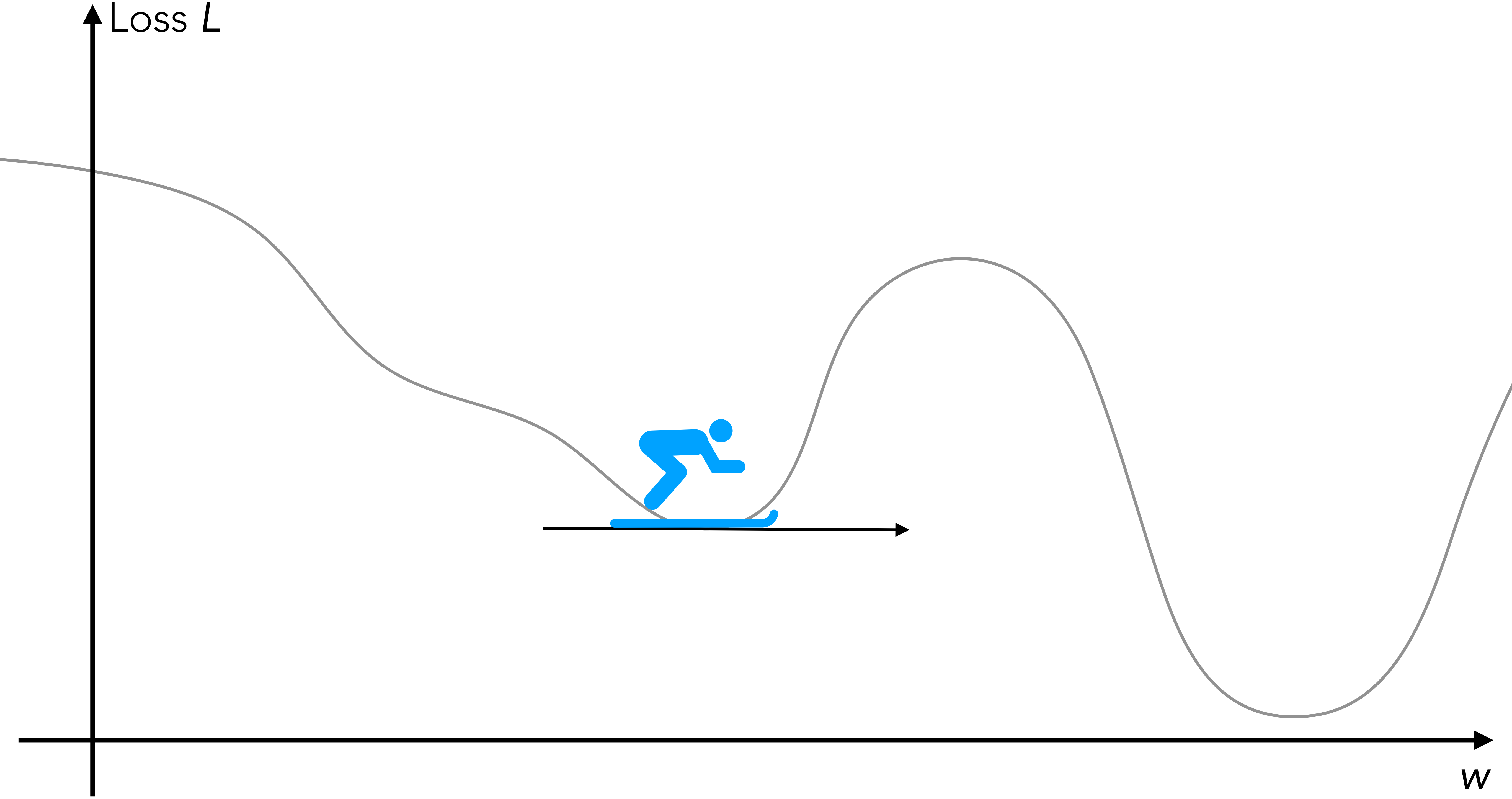
Learning a Deep Net

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



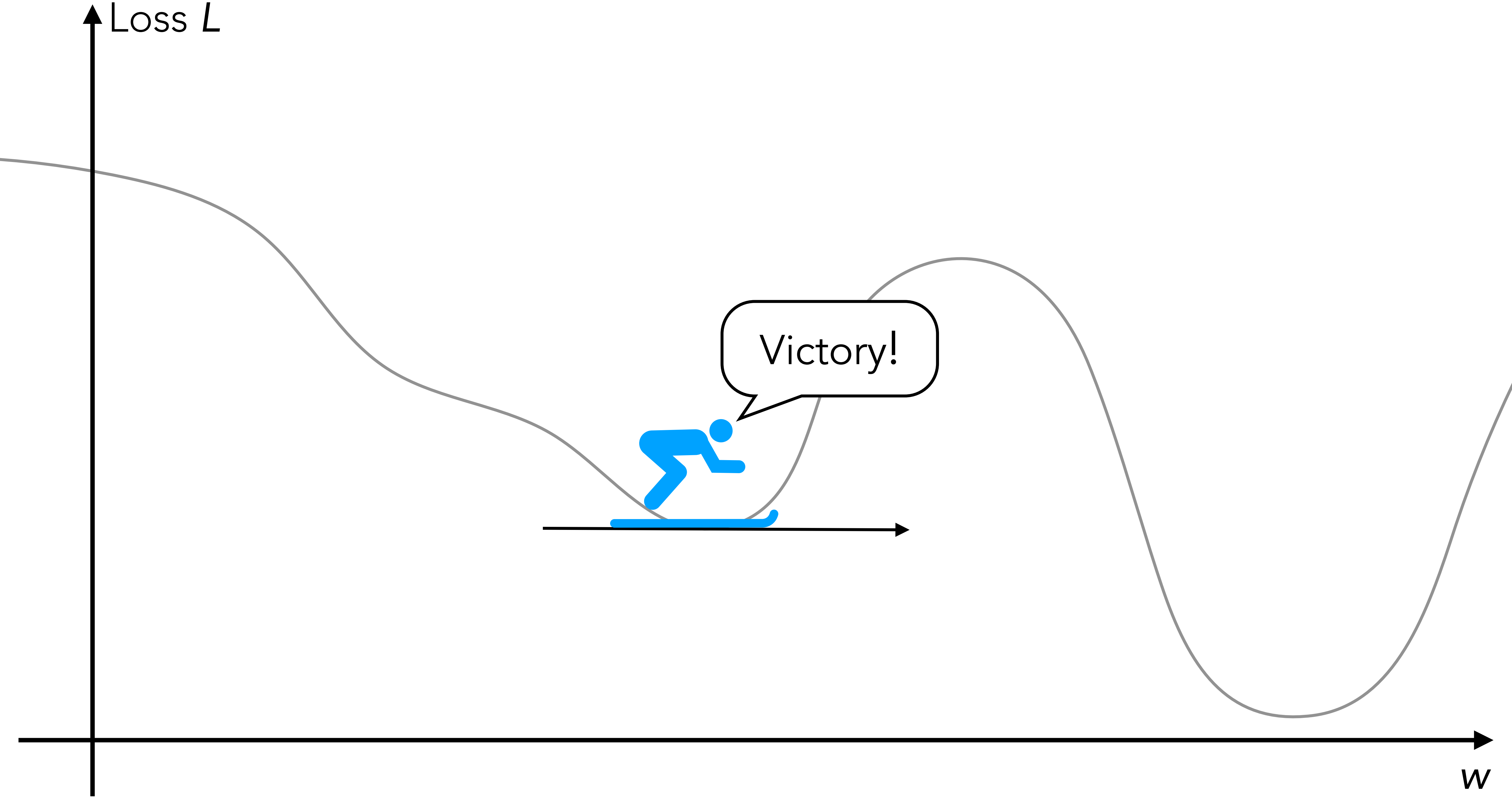
Learning a Deep Net

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



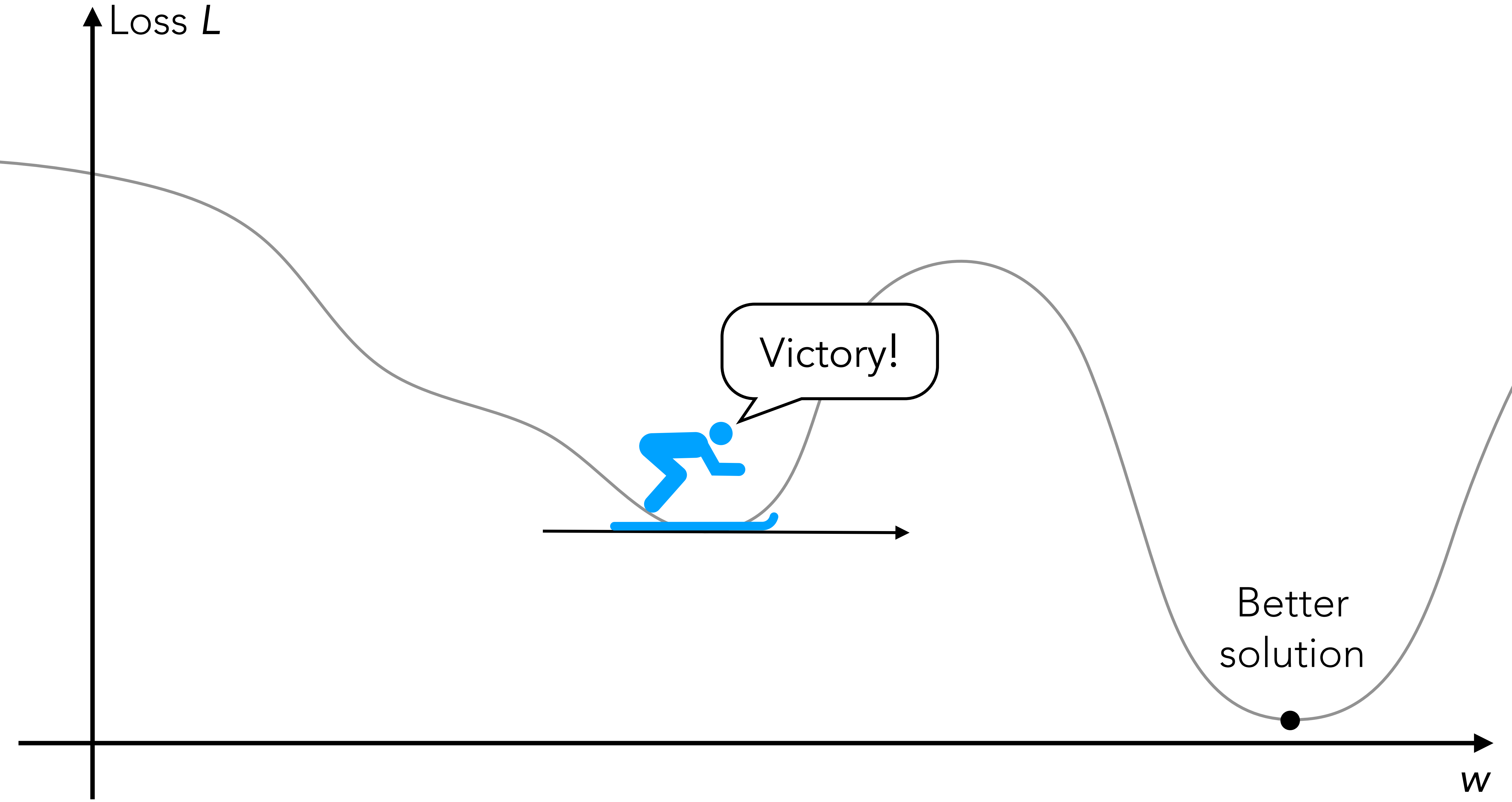
Learning a Deep Net

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



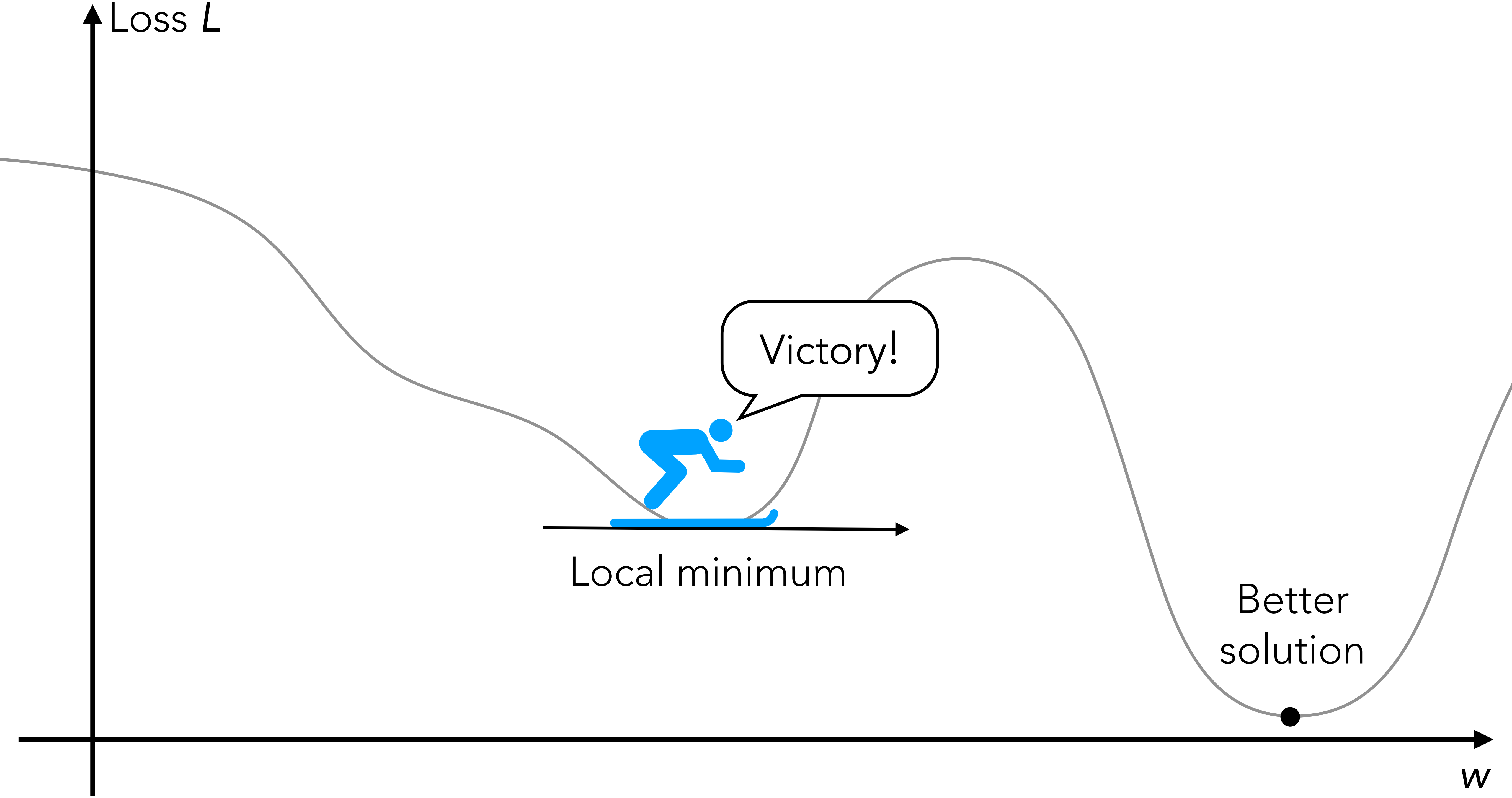
Learning a Deep Net

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



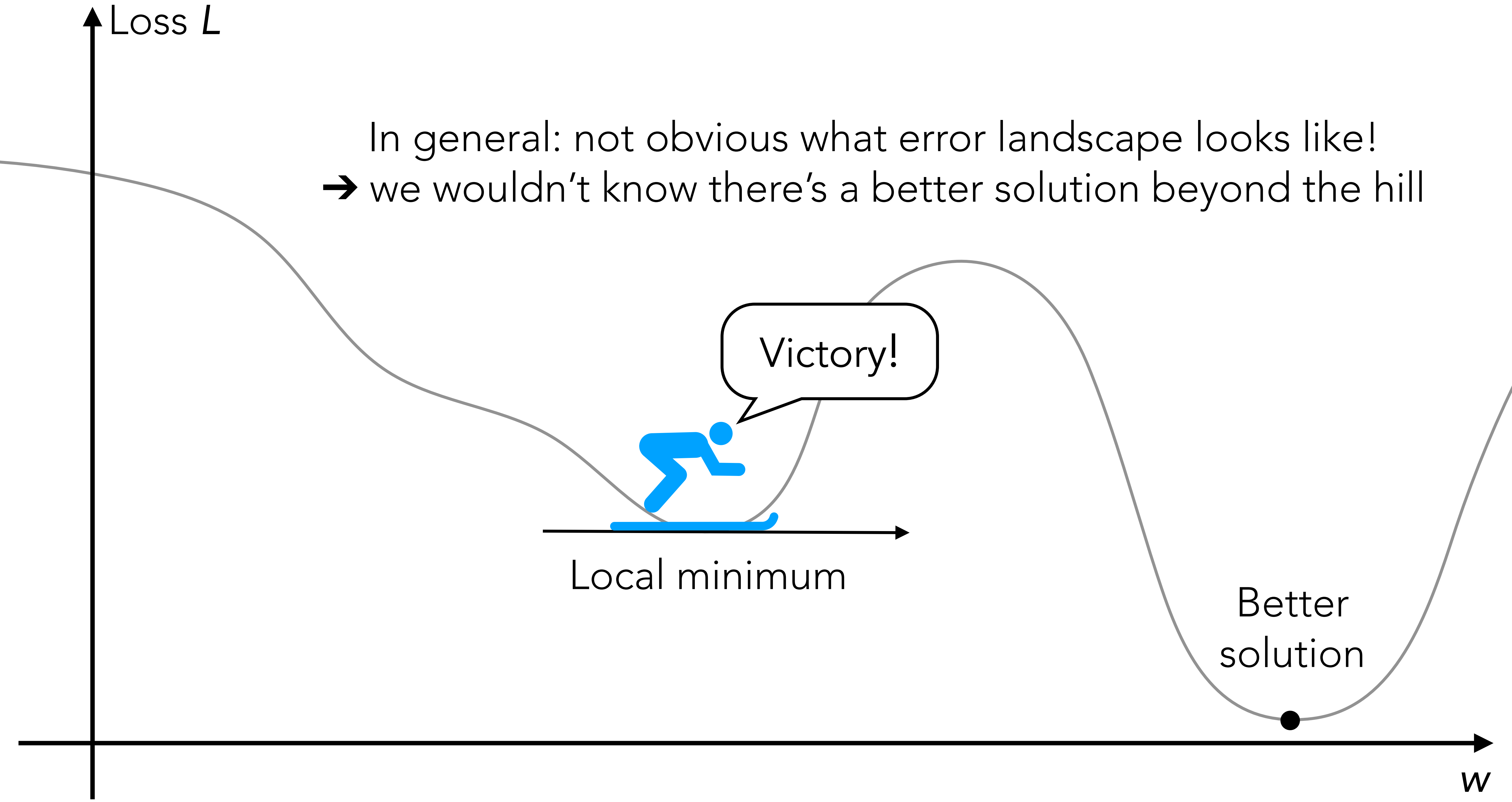
Learning a Deep Net

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



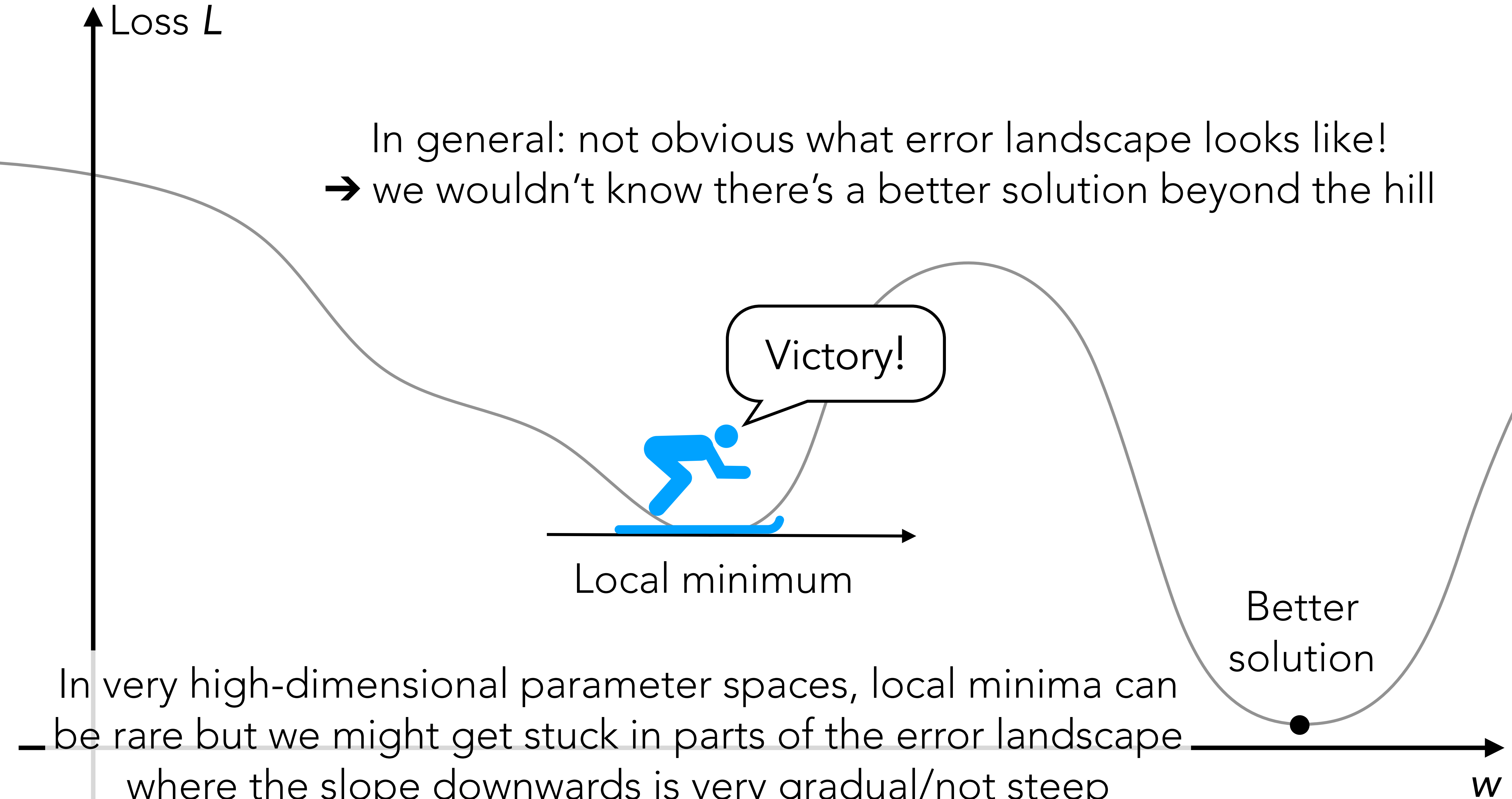
Learning a Deep Net

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



Learning a Deep Net

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

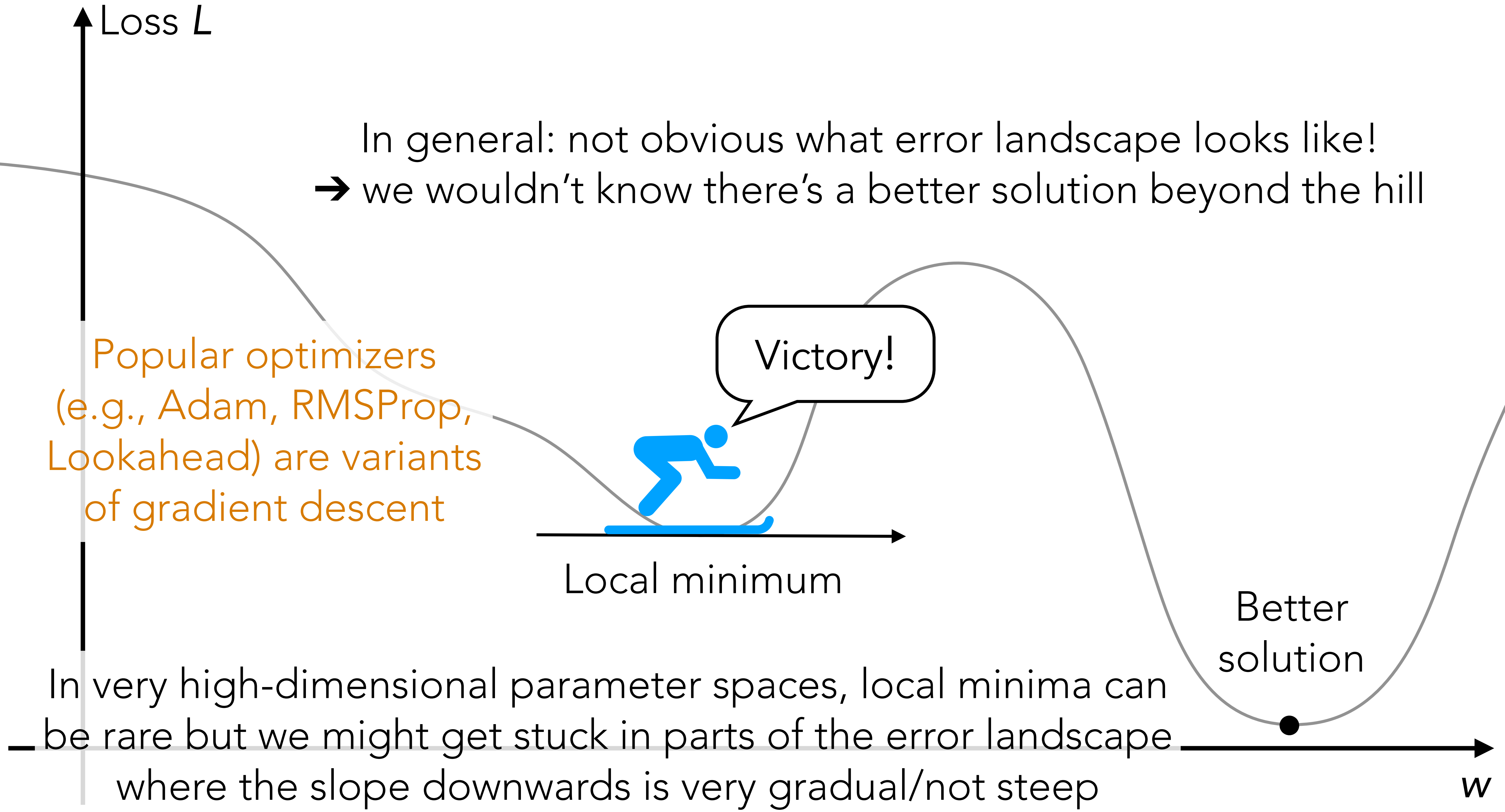


In general: not obvious what error landscape looks like!
➔ we wouldn't know there's a better solution beyond the hill

In very high-dimensional parameter spaces, local minima can be rare but we might get stuck in parts of the error landscape where the slope downwards is very gradual/not steep

Learning a Deep Net

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



In general: not obvious what error landscape looks like!
➔ we wouldn't know there's a better solution beyond the hill

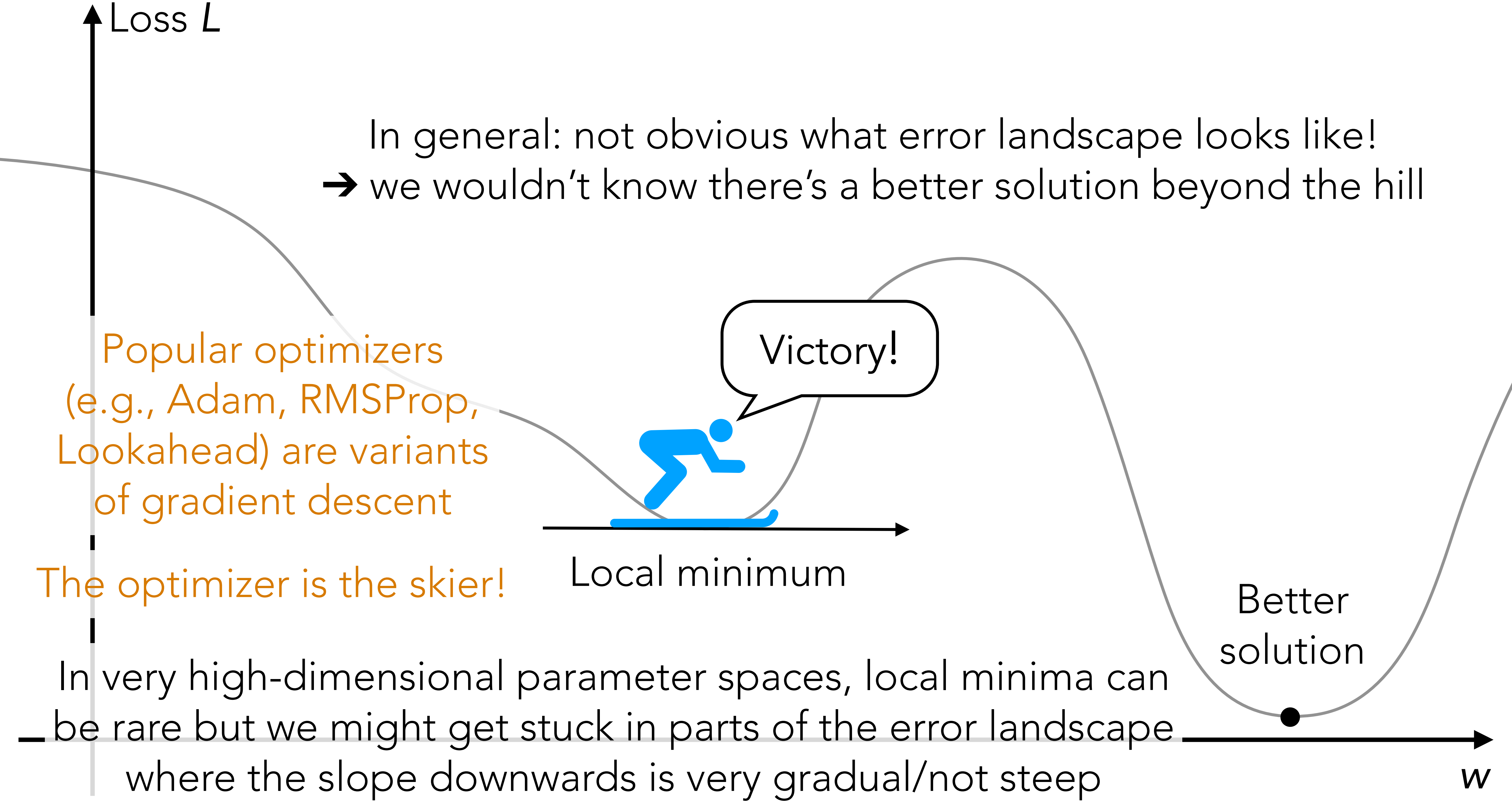
Popular optimizers (e.g., Adam, RMSProp, Lookahead) are variants of gradient descent

In very high-dimensional parameter spaces, local minima can be rare but we might get stuck in parts of the error landscape where the slope downwards is very gradual/not steep

Learning a Deep Net

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

In general: not obvious what error landscape looks like!
➔ we wouldn't know there's a better solution beyond the hill

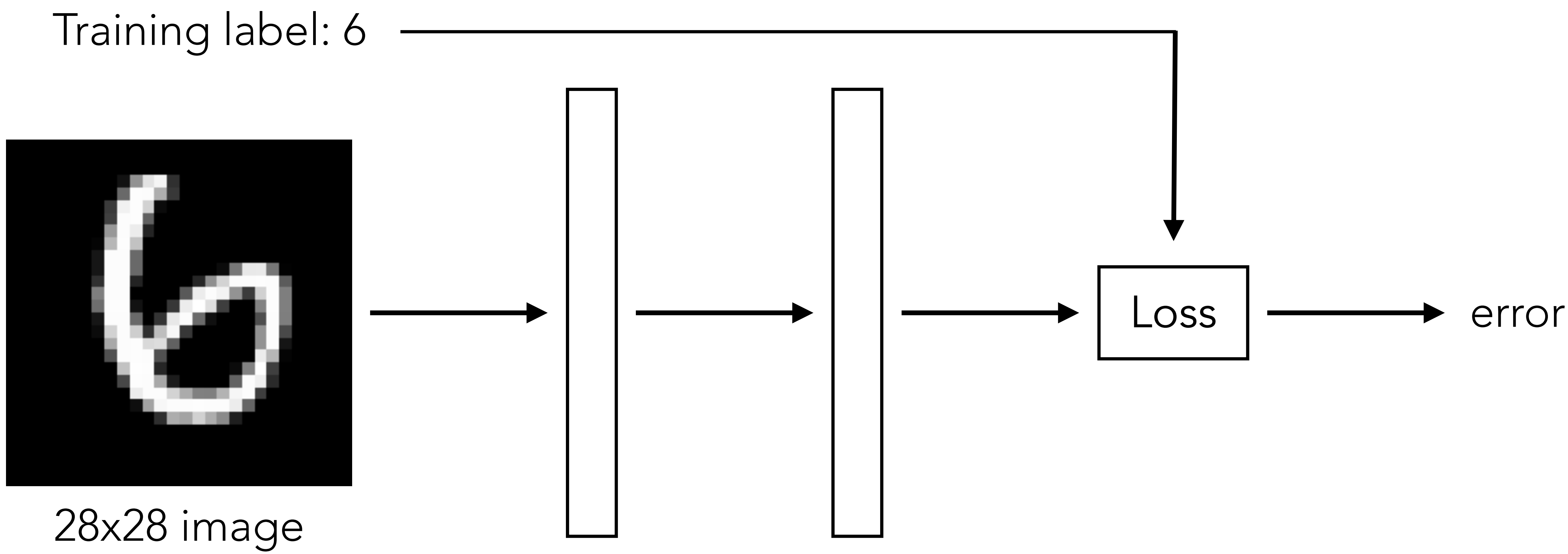


Popular optimizers (e.g., Adam, RMSProp, Lookahead) are variants of gradient descent

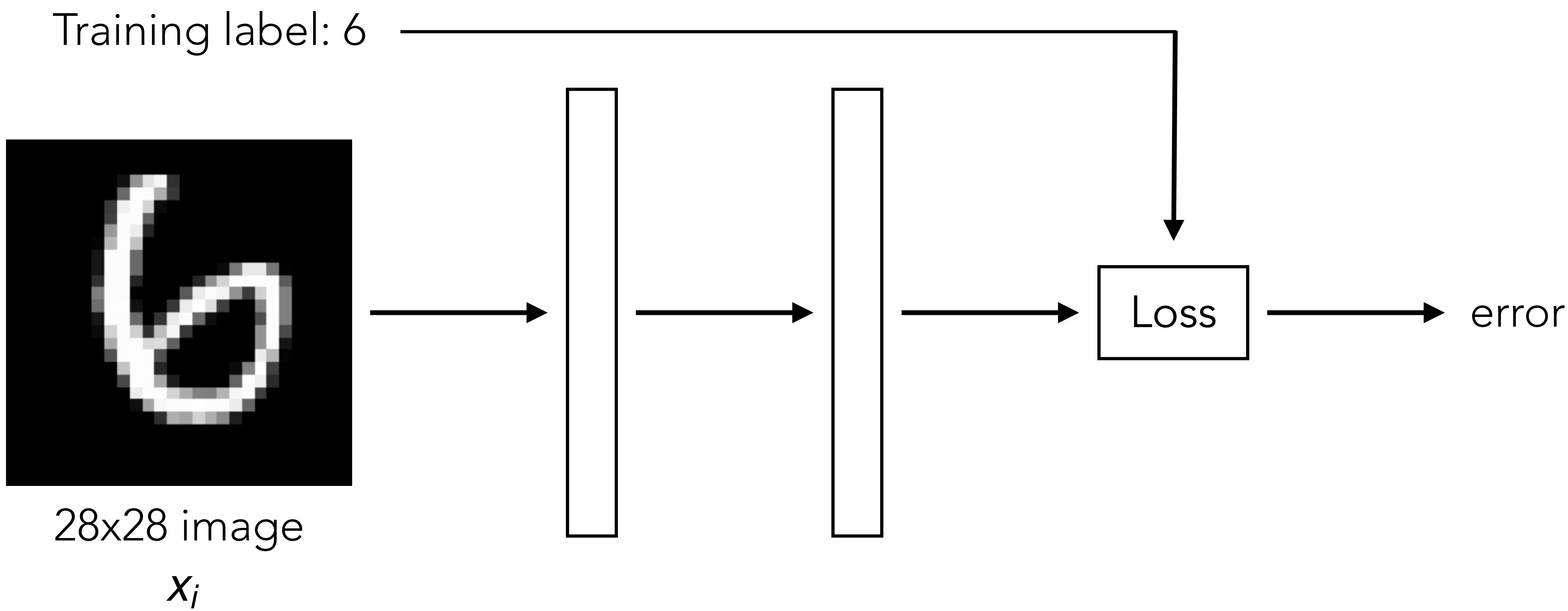
The optimizer is the skier!

In very high-dimensional parameter spaces, local minima can be rare but we might get stuck in parts of the error landscape where the slope downwards is very gradual/not steep

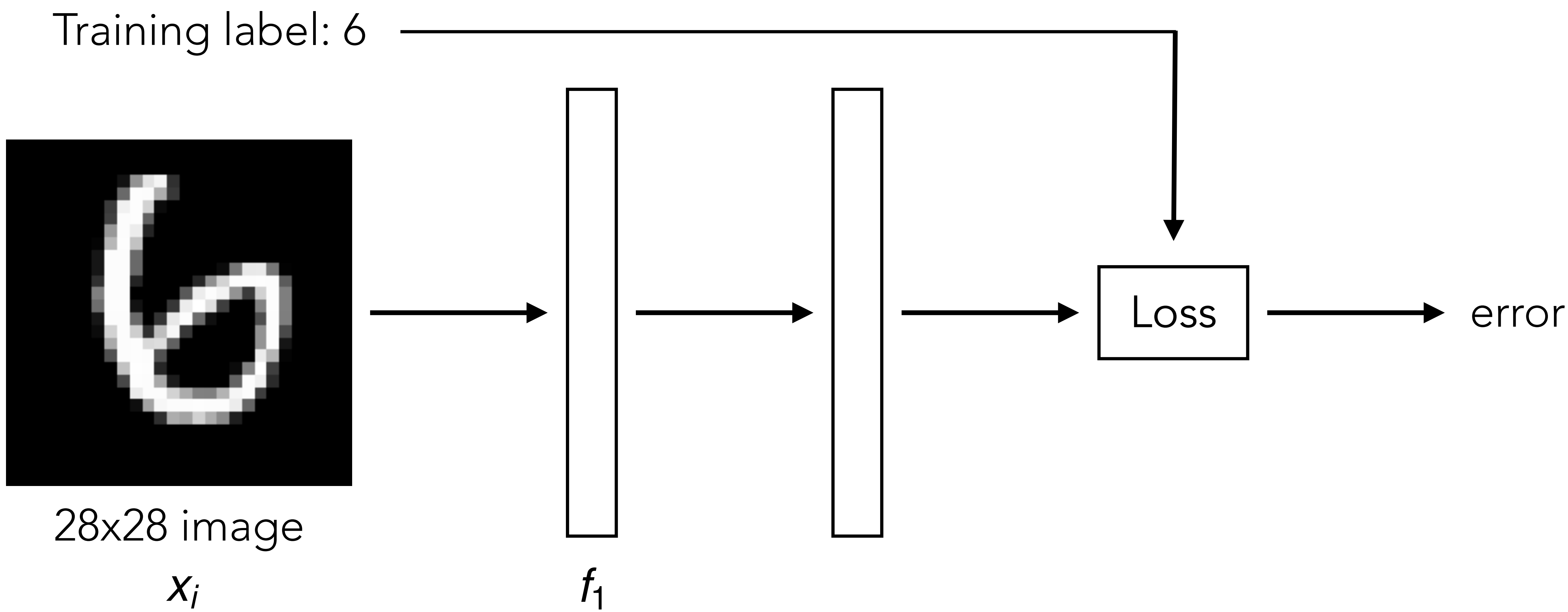
Handwritten Digit Recognition



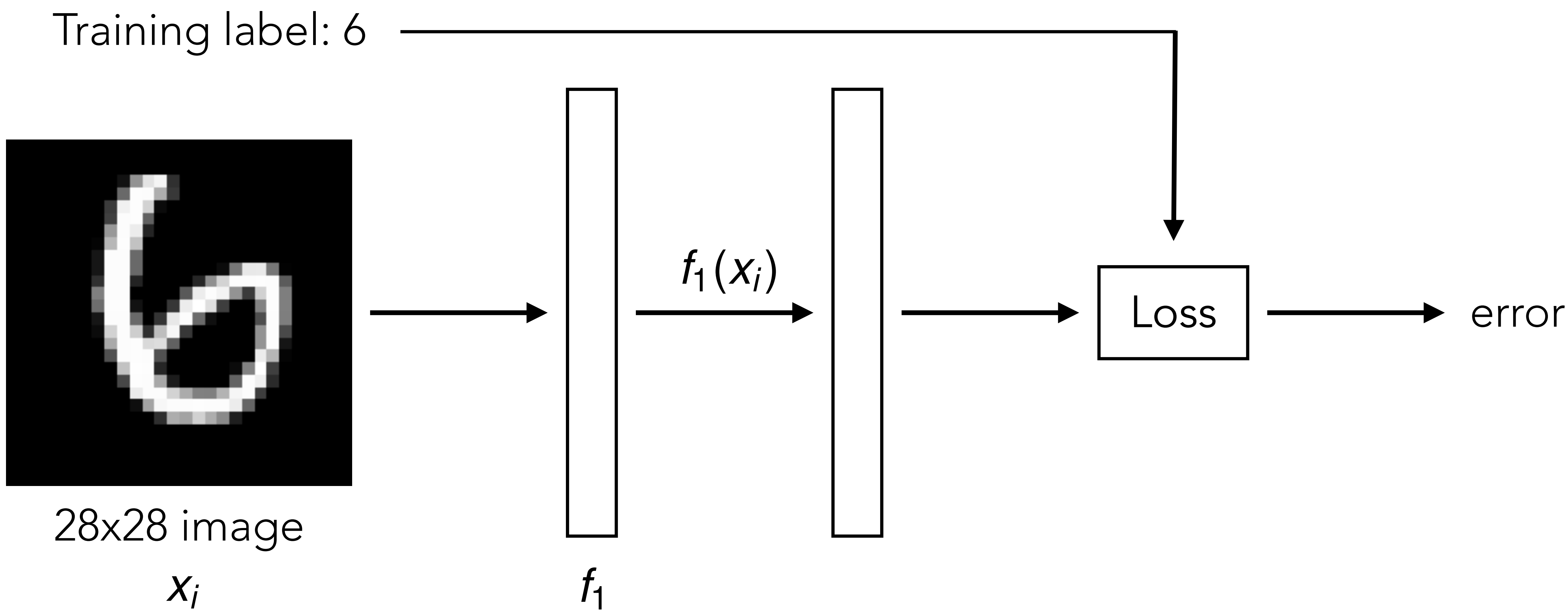
Handwritten Digit Recognition



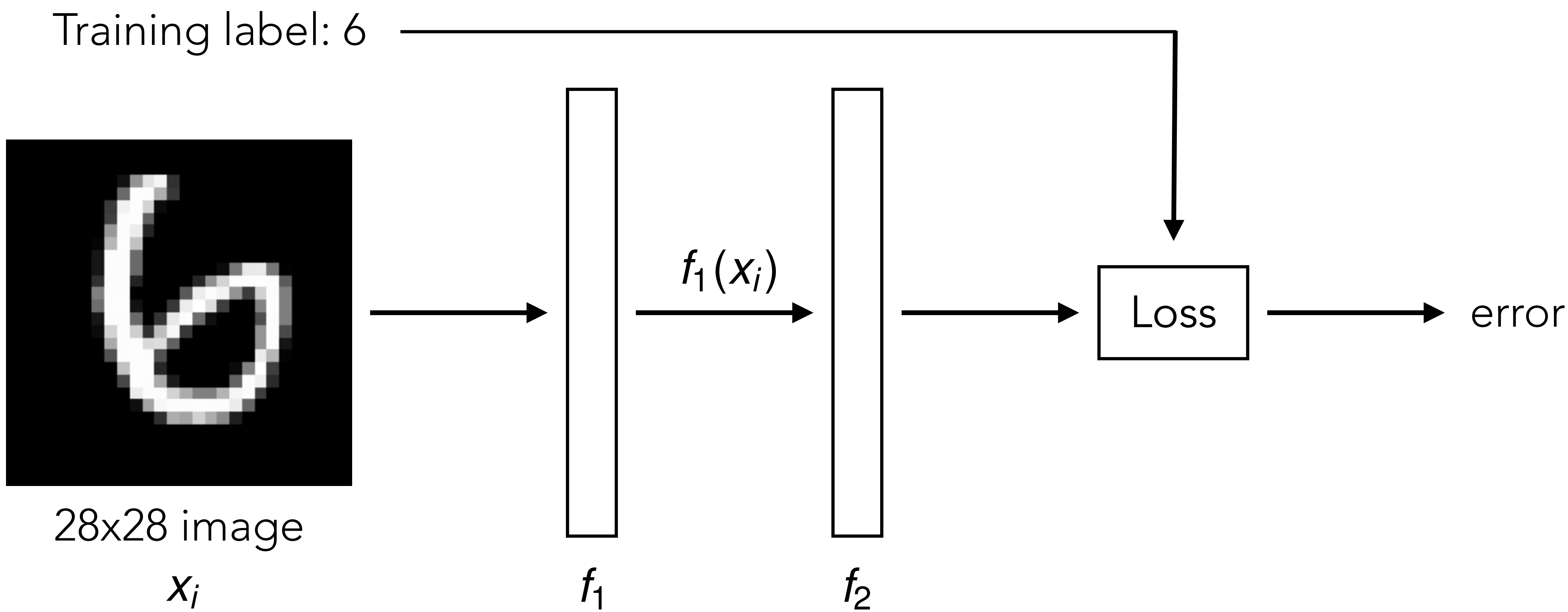
Handwritten Digit Recognition



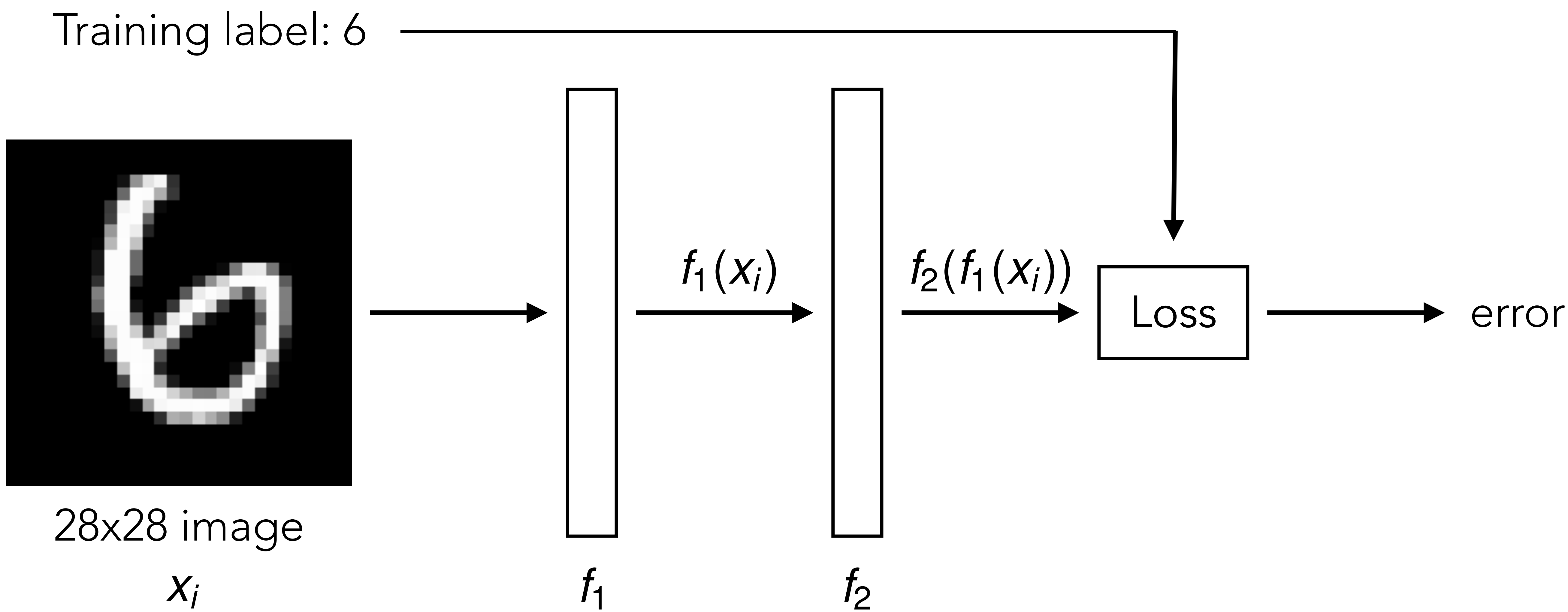
Handwritten Digit Recognition



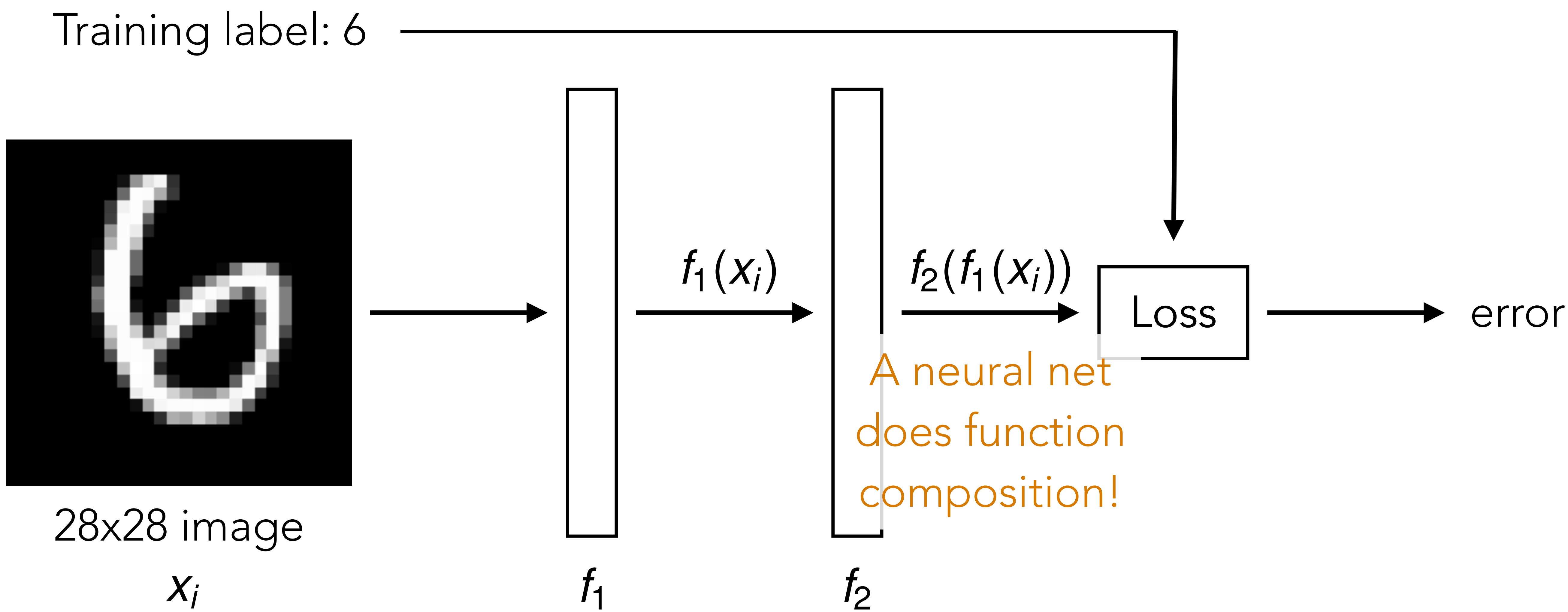
Handwritten Digit Recognition



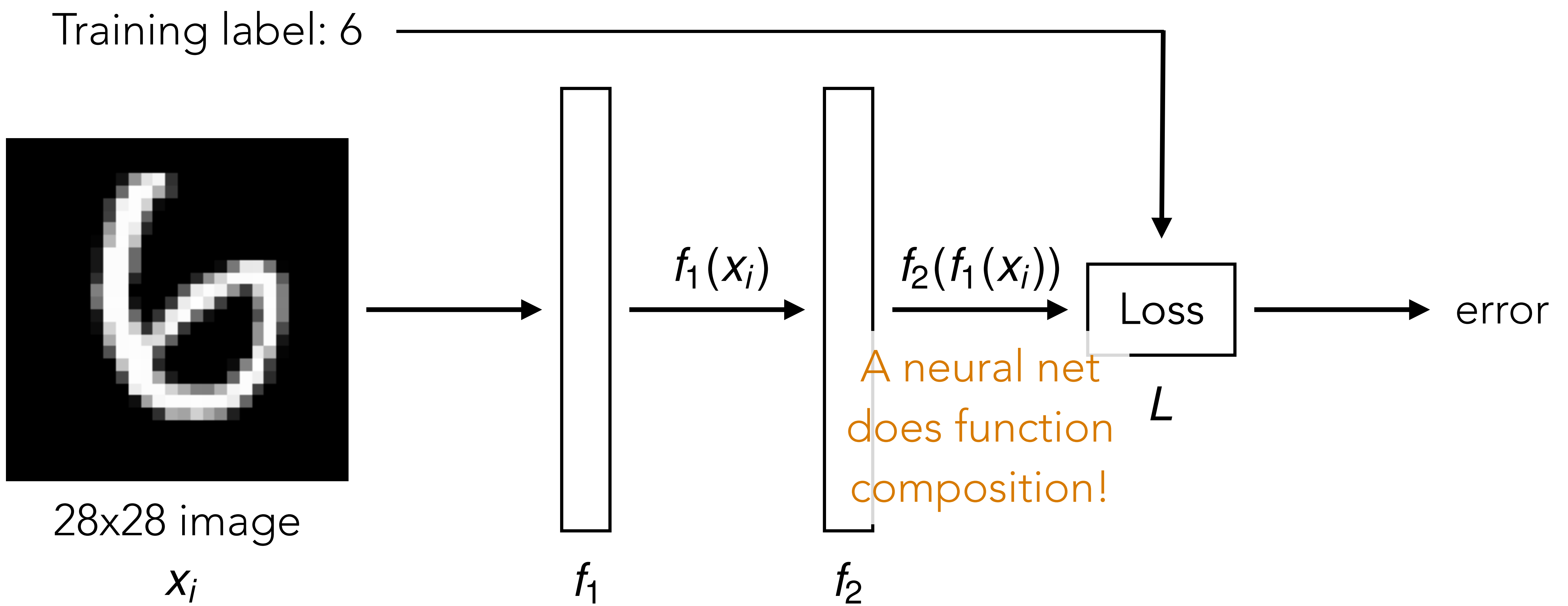
Handwritten Digit Recognition



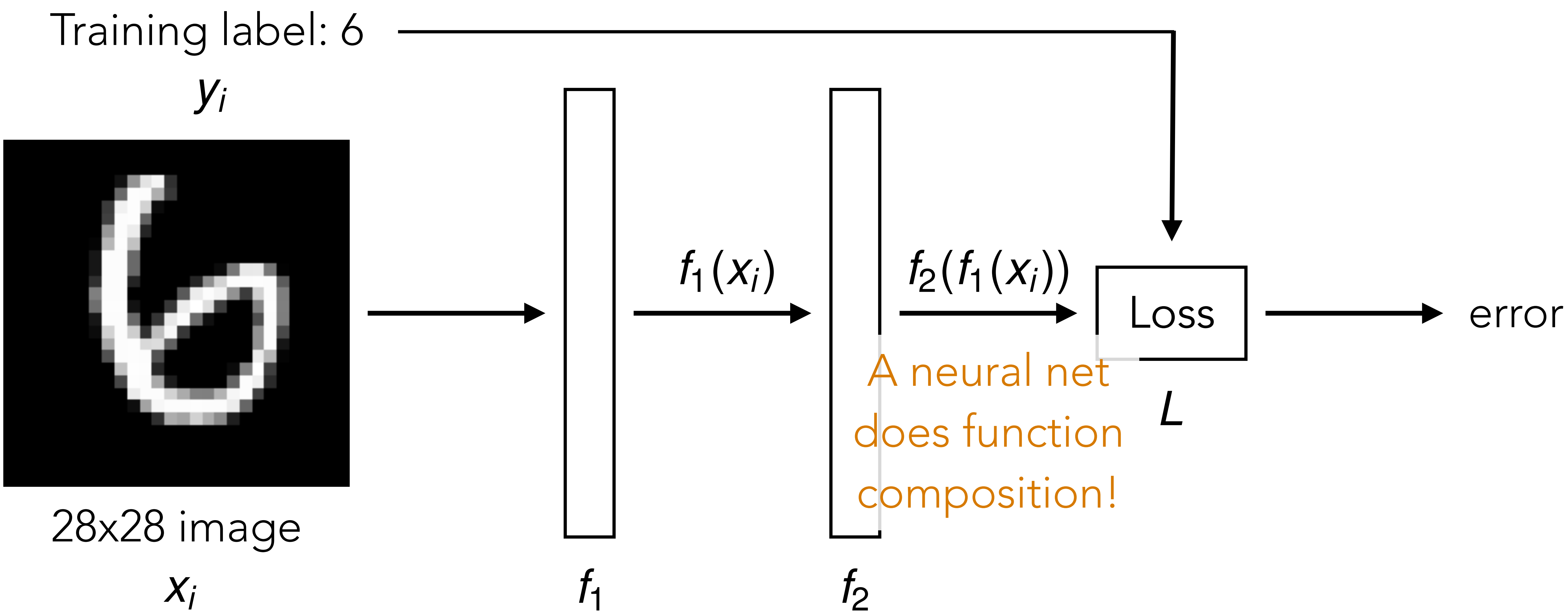
Handwritten Digit Recognition



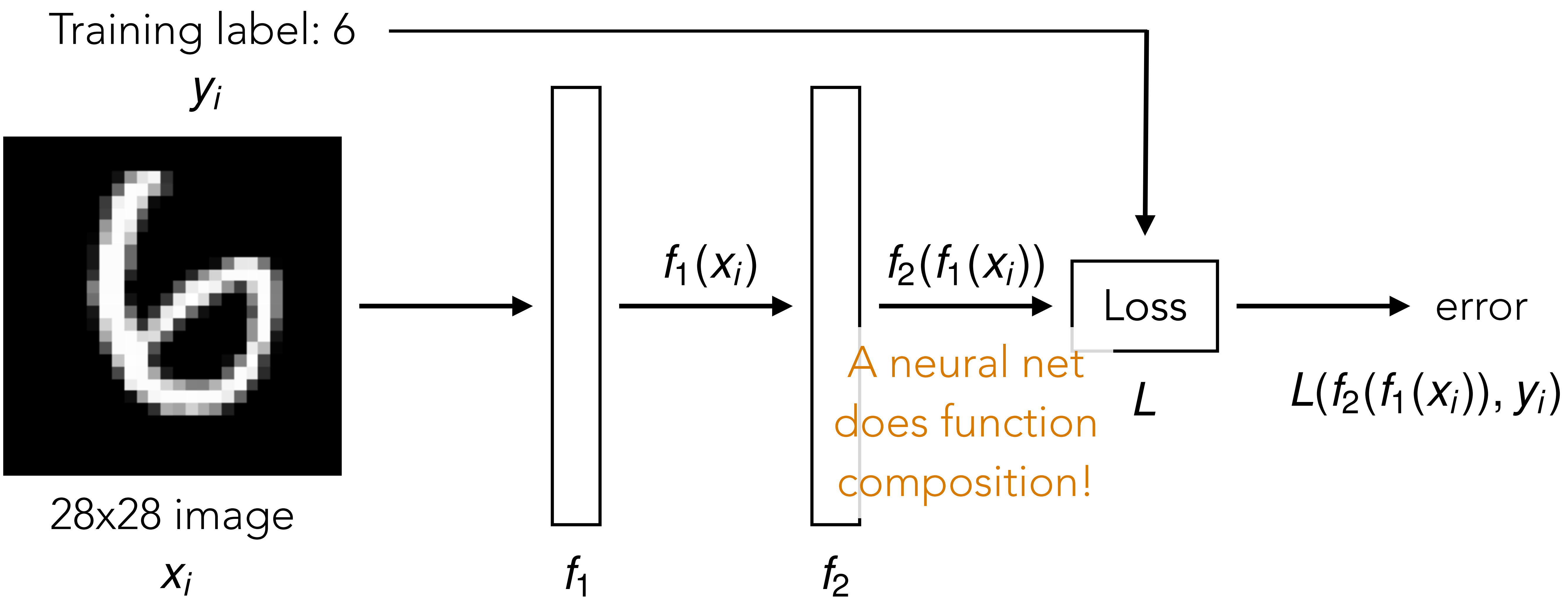
Handwritten Digit Recognition



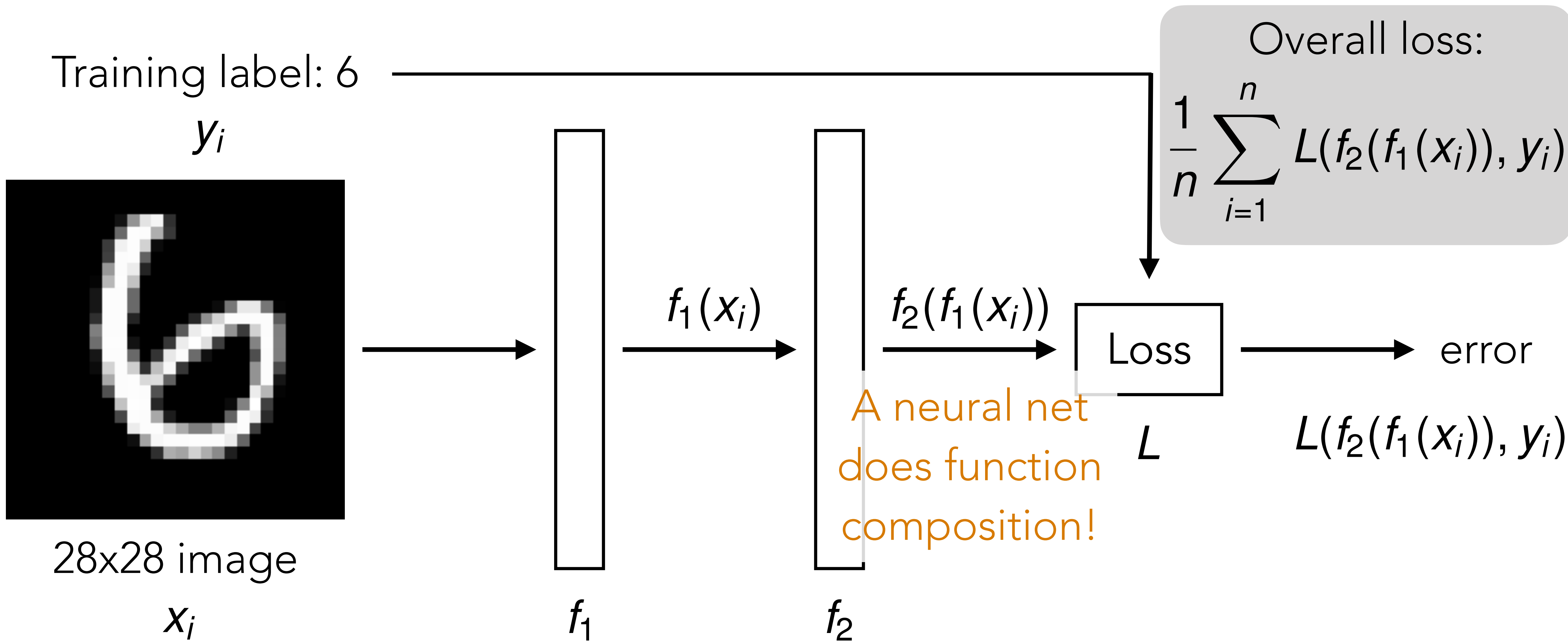
Handwritten Digit Recognition



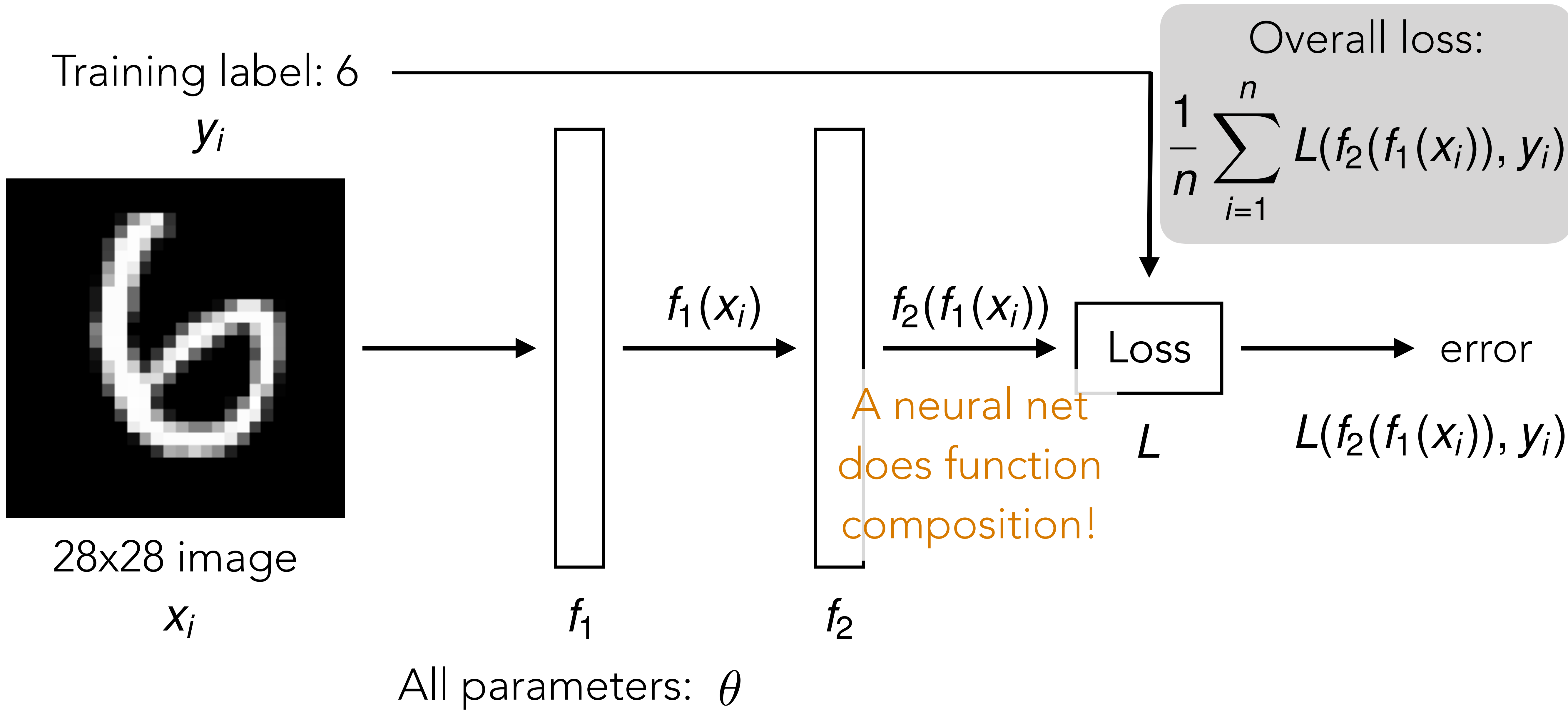
Handwritten Digit Recognition



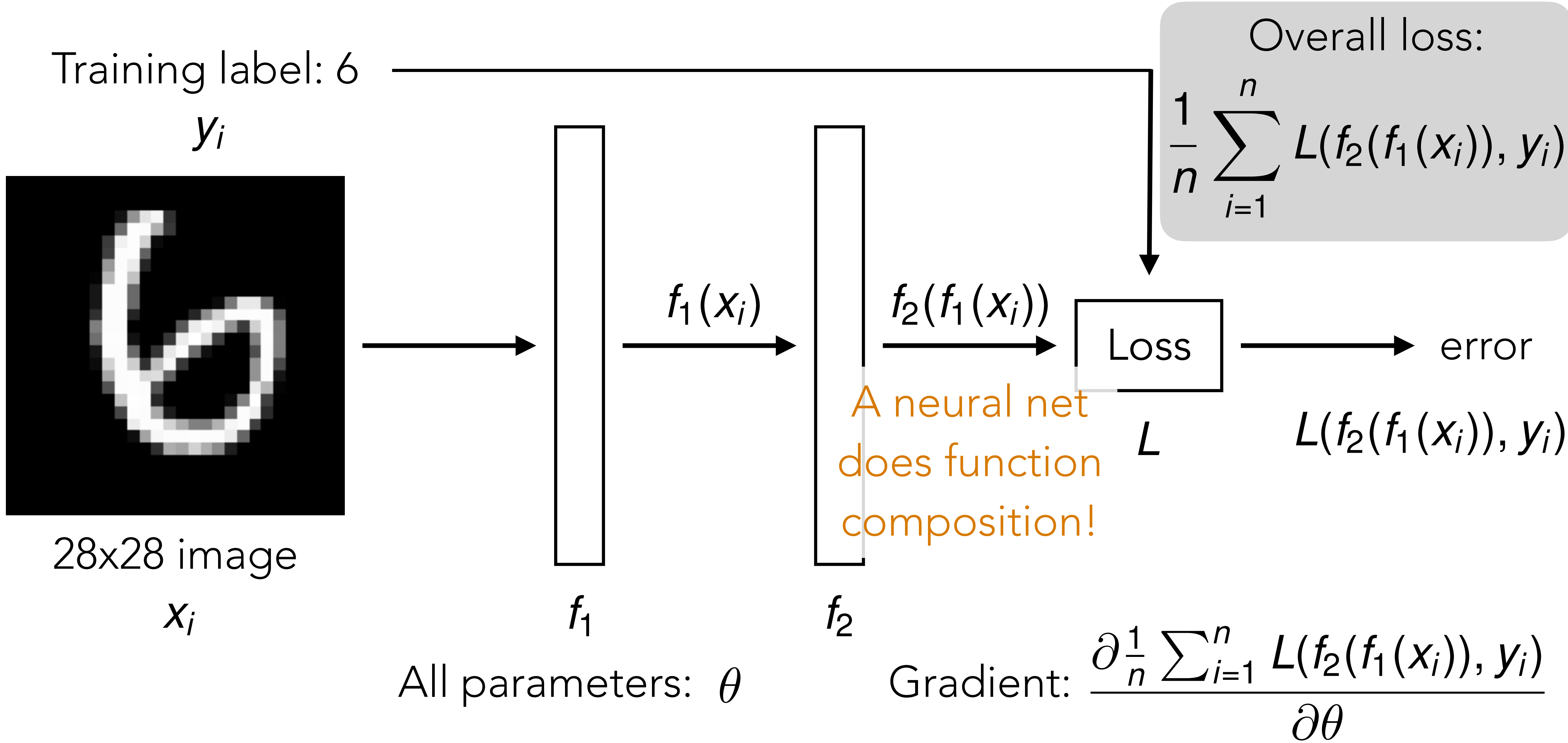
Handwritten Digit Recognition



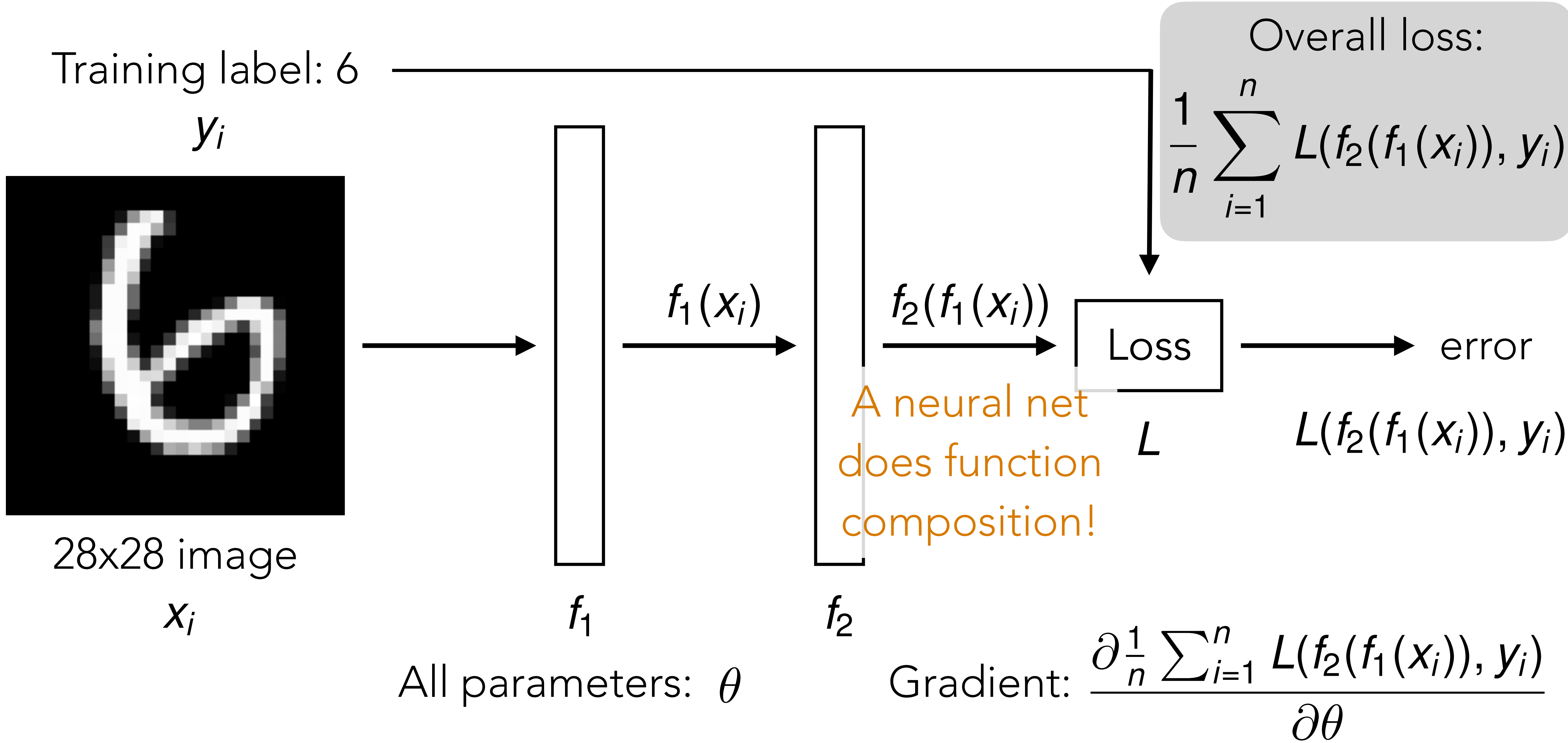
Handwritten Digit Recognition



Handwritten Digit Recognition

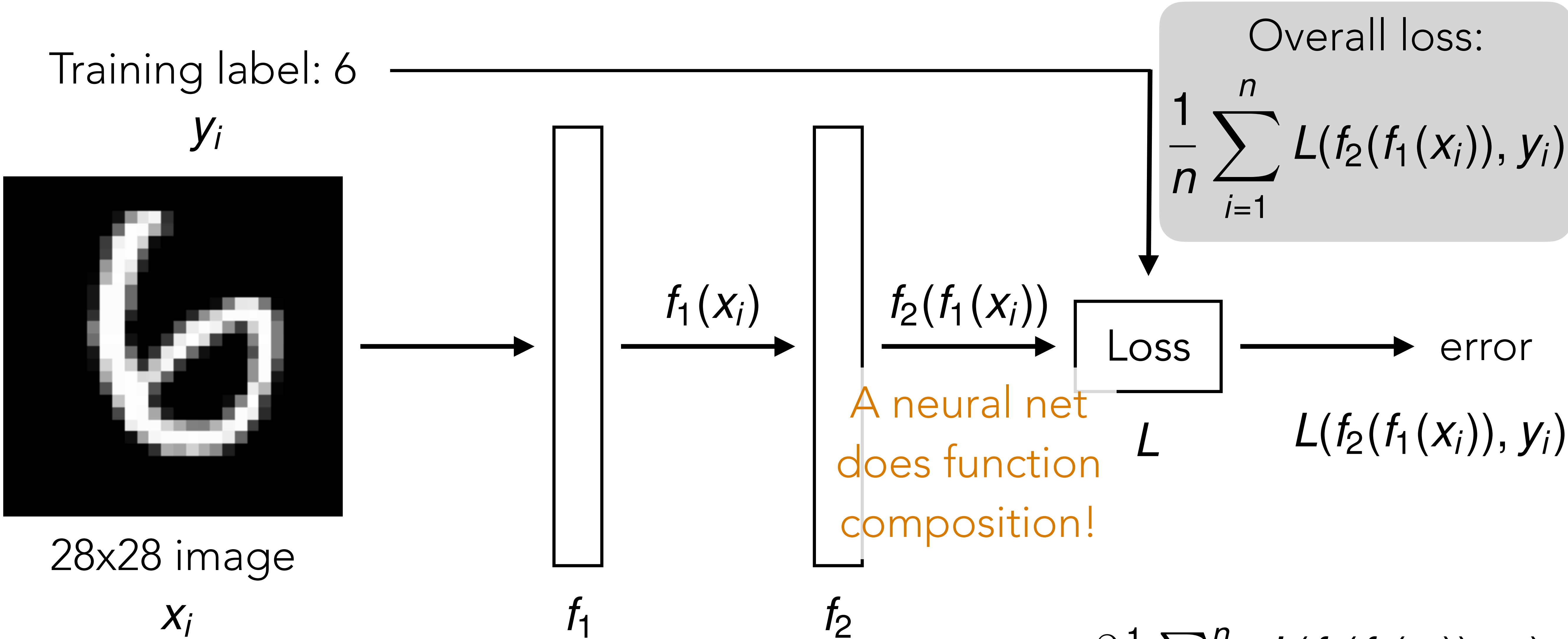


Handwritten Digit Recognition



Automatic differentiation is crucial in learning deep nets!

Handwritten Digit Recognition



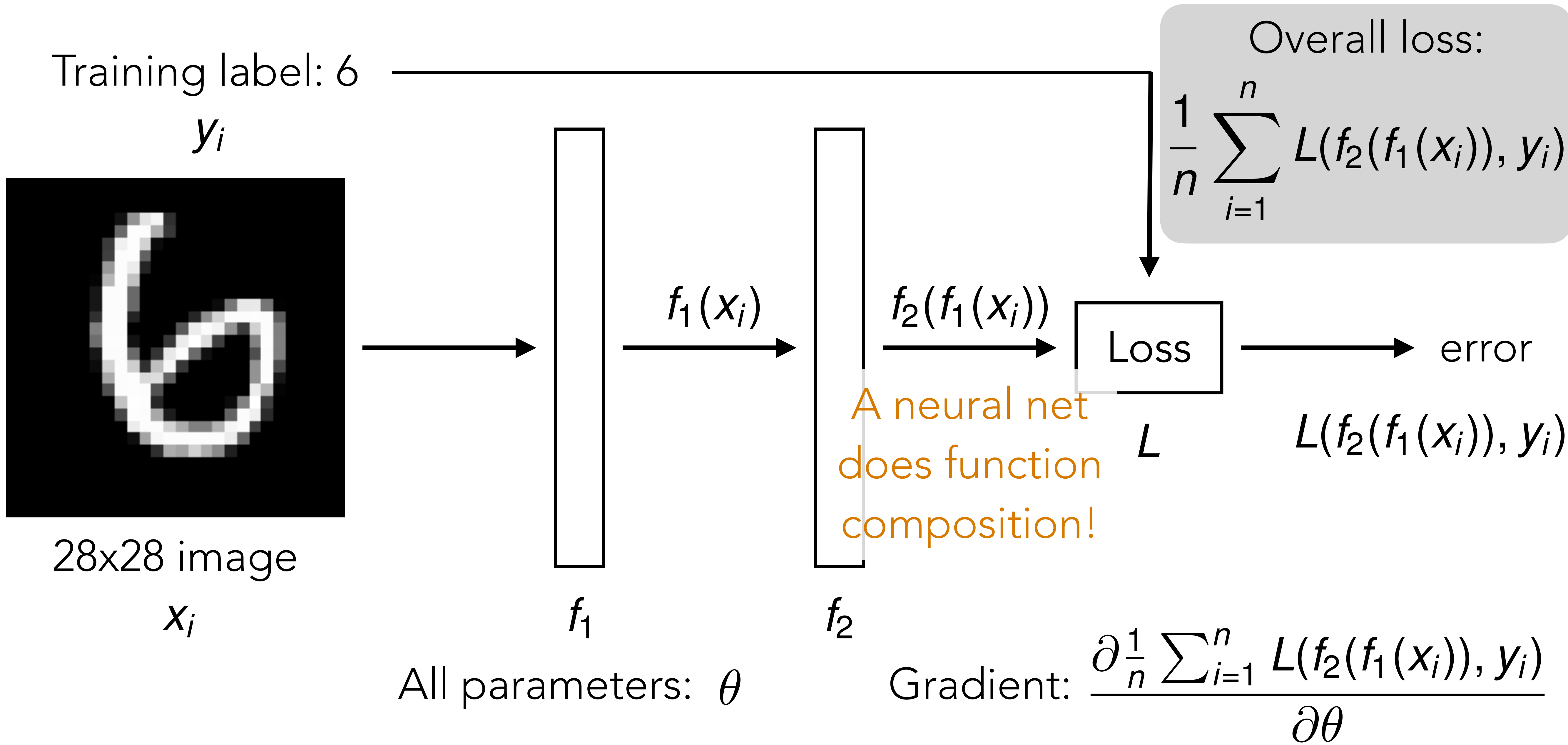
All parameters: θ

Gradient: $\frac{\partial \frac{1}{n}\sum_{i=1}^{n} L(f_2(f_1(x_i)), y_i)}{\partial \theta}$

Automatic differentiation is crucial in learning deep nets!

Taking the derivative of a function composition is done using the **chain rule**

Handwritten Digit Recognition



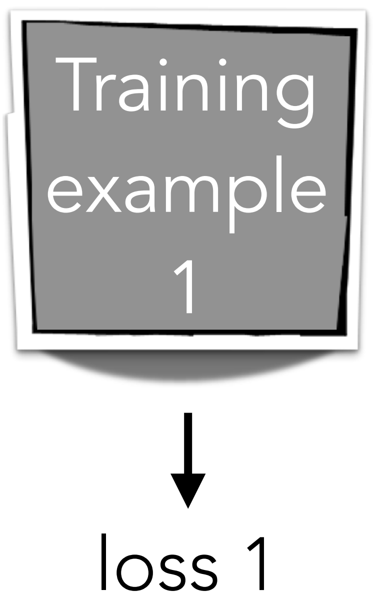
Automatic differentiation is crucial in learning deep nets!

Taking the derivative of a function composition is done using the **chain rule**

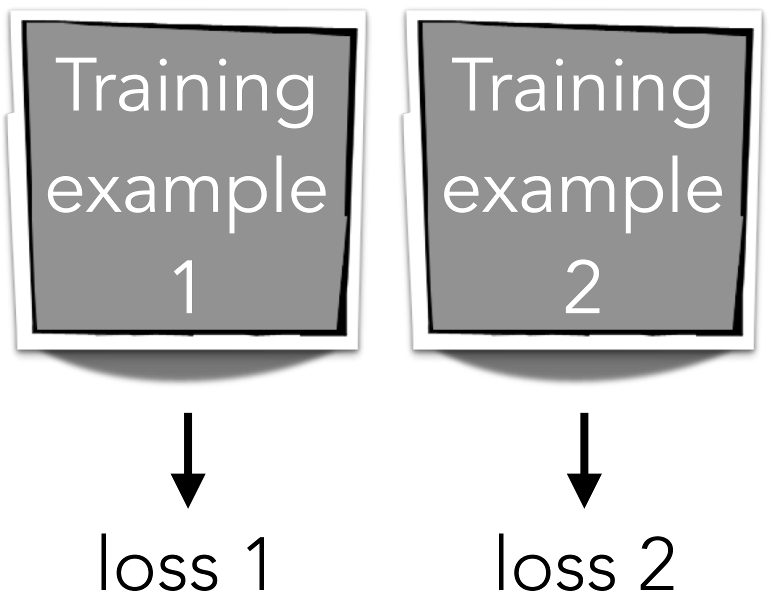
Algorithm to compute the gradient using the chain rule: **back-propagation**

Gradient Descent

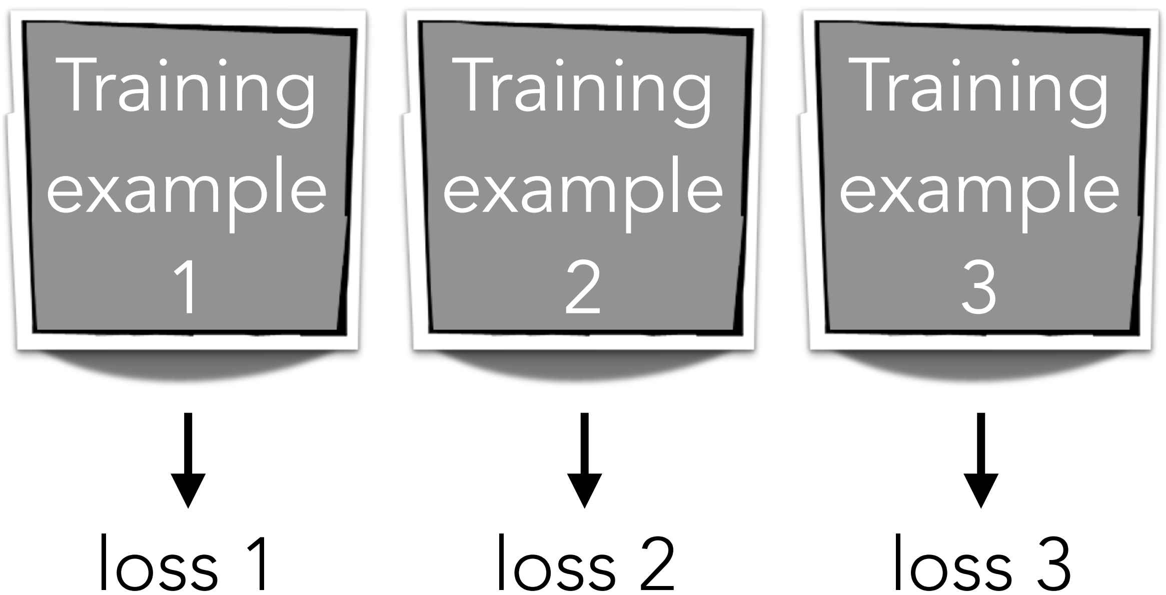
Gradient Descent



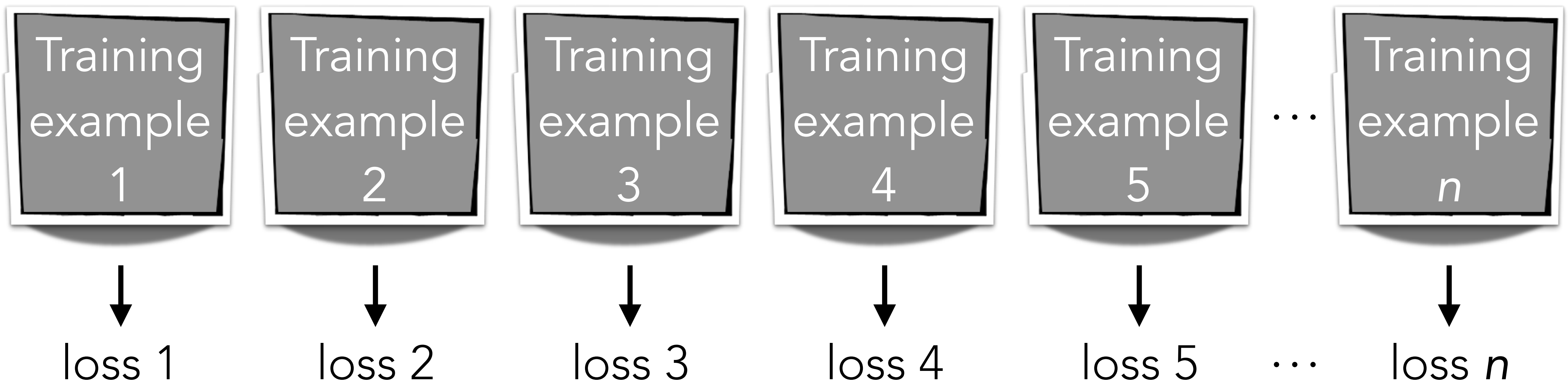
Gradient Descent



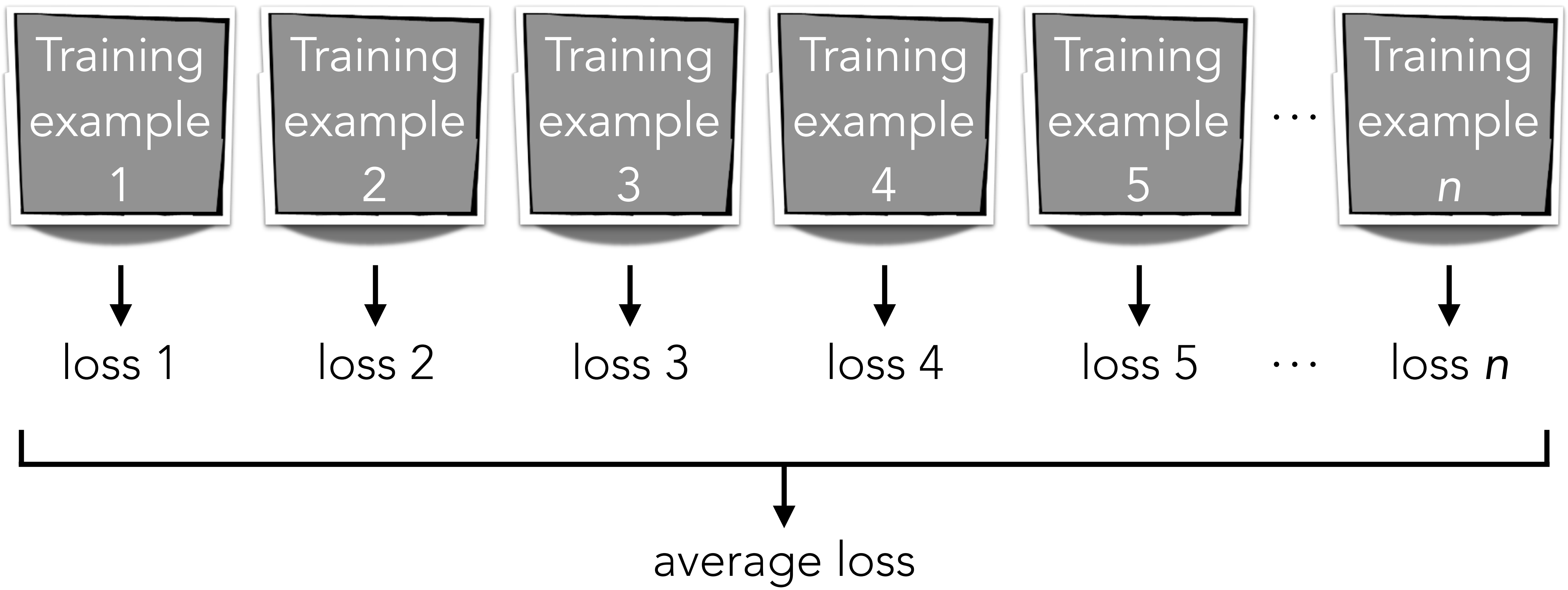
Gradient Descent



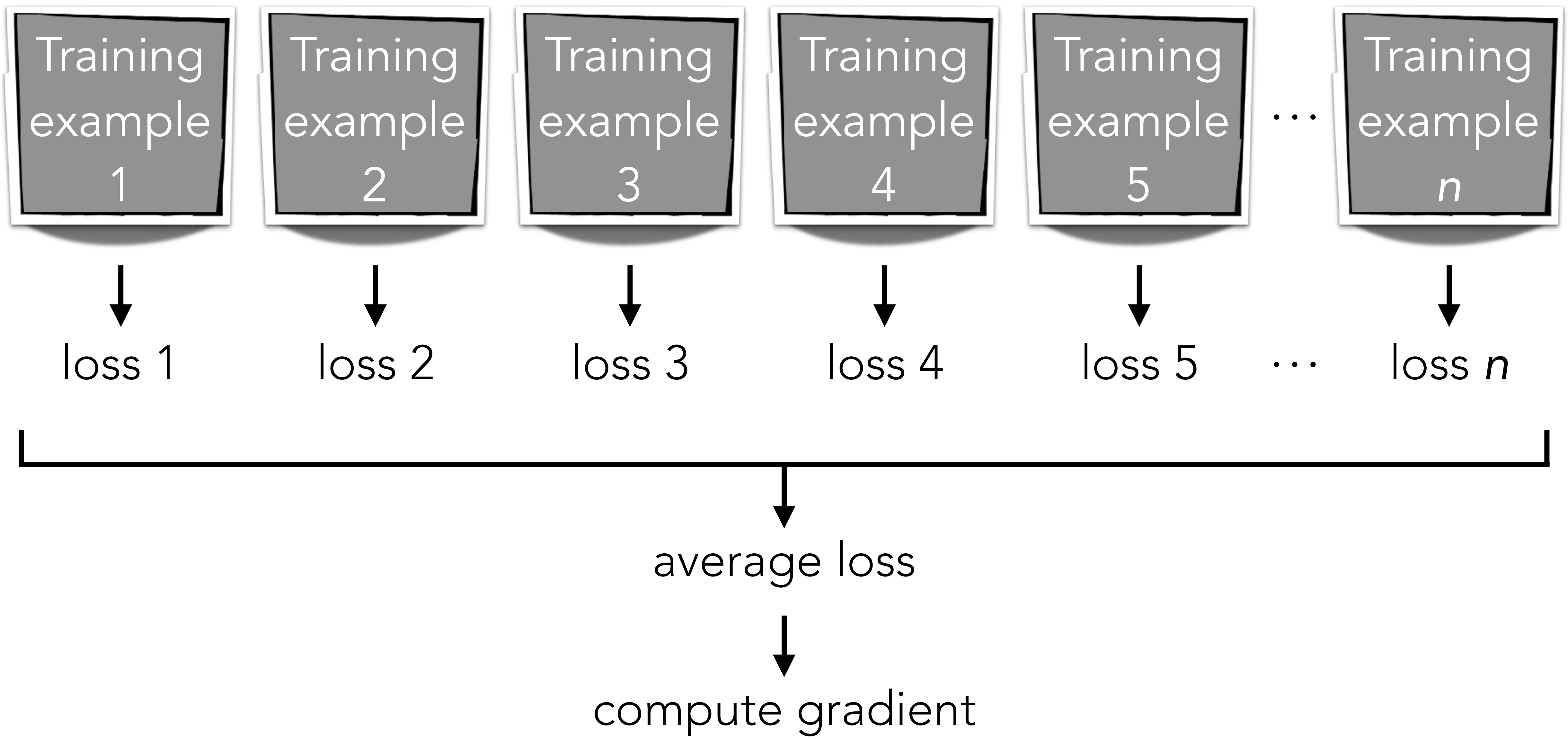
Gradient Descent



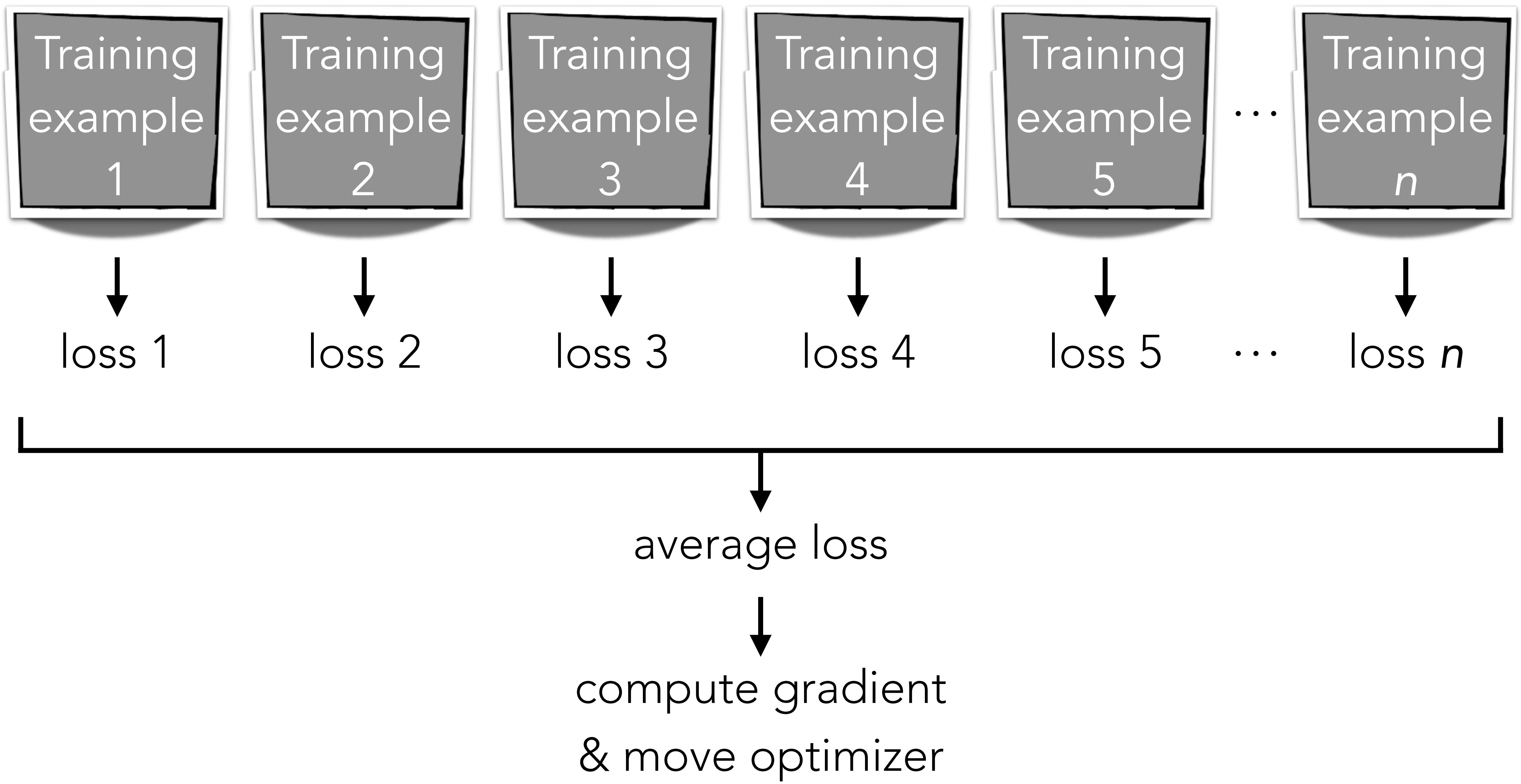
Gradient Descent



Gradient Descent



Gradient Descent



Gradient Descent

Training example 1 → loss 1

Training example 2 → loss 2

Training example 3 → loss 3

Training example 4 → loss 4

Training example 5 → loss 5

…

Training example *n* → loss *n*

↓

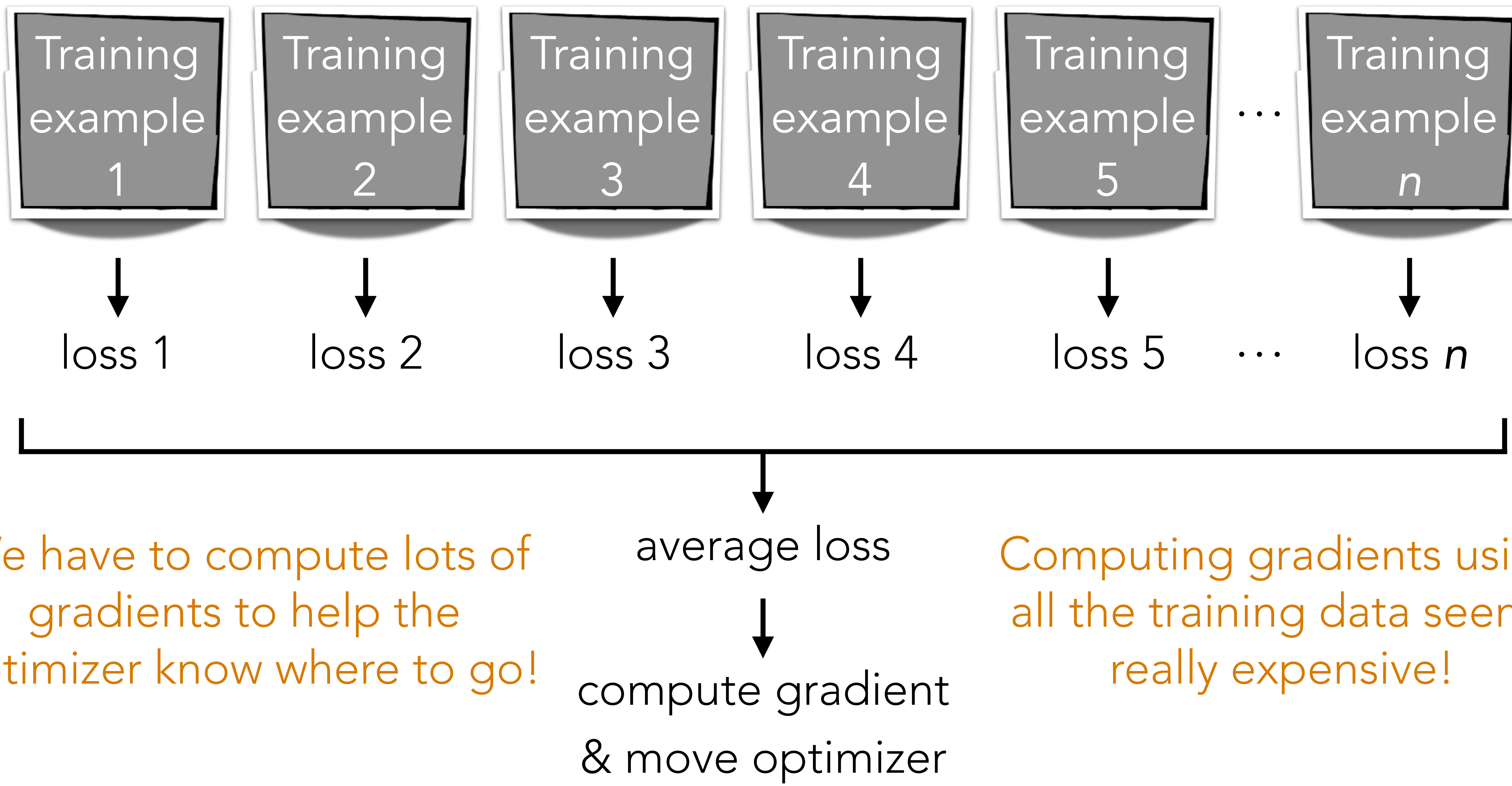
average loss

↓

compute gradient & move optimizer

We have to compute lots of gradients to help the optimizer know where to go!

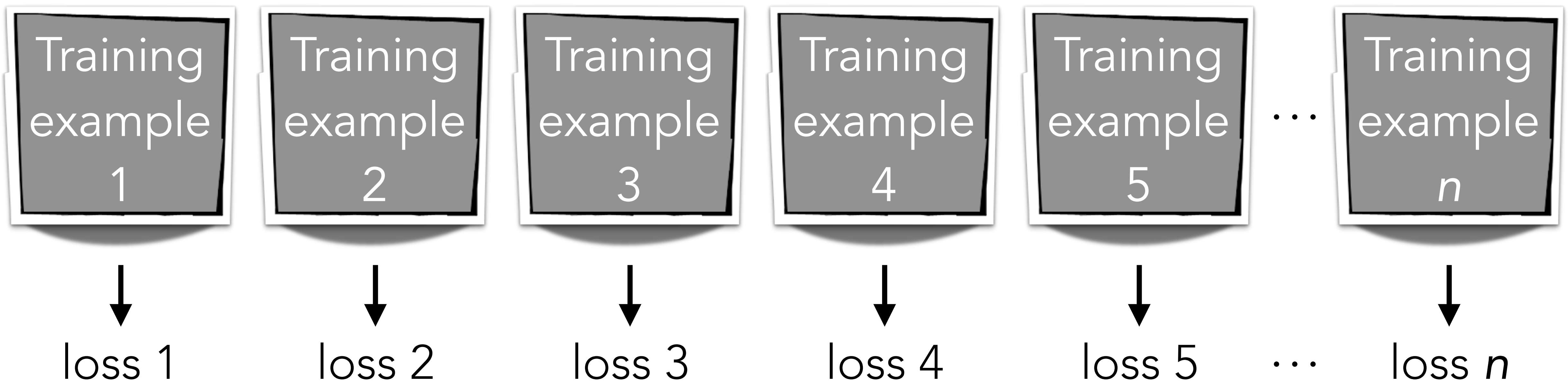
Gradient Descent



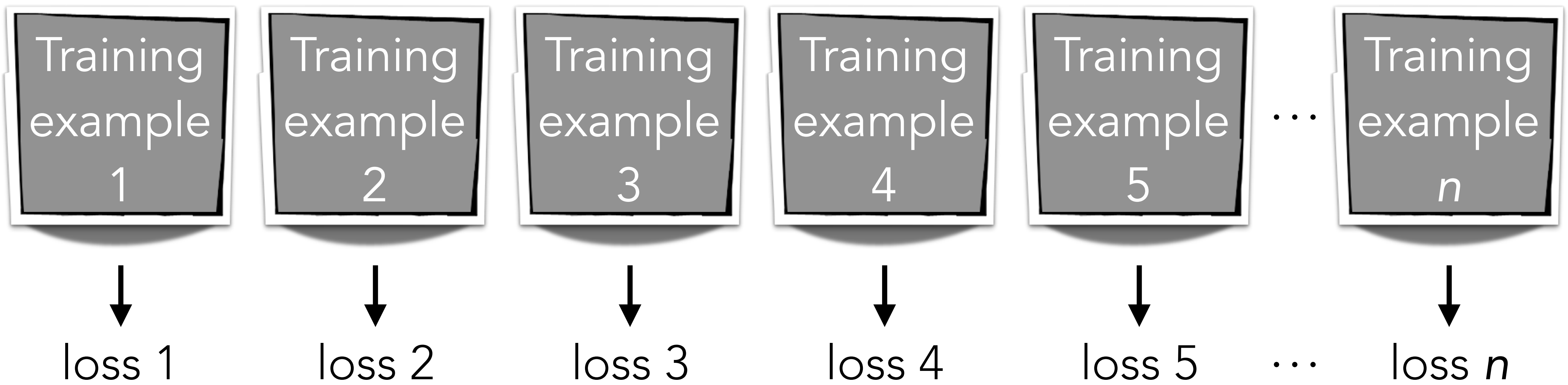
We have to compute lots of gradients to help the optimizer know where to go!

Computing gradients using all the training data seems really expensive!

Stochastic Gradient Descent (SGD)

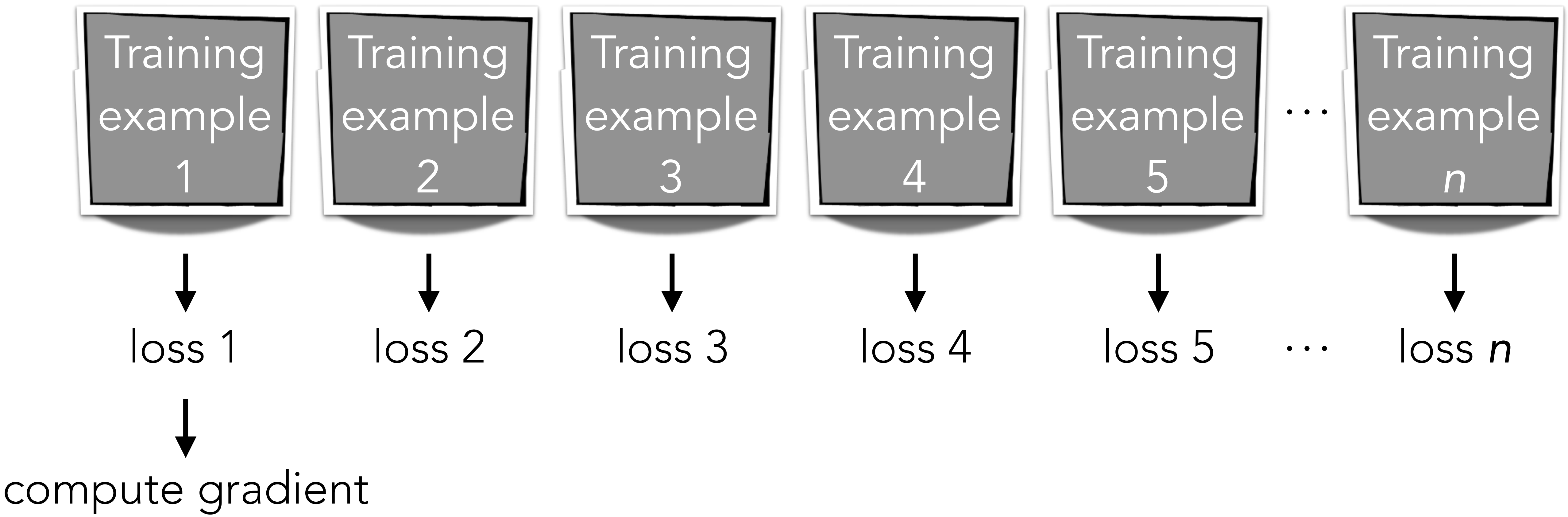


Stochastic Gradient Descent (SGD)



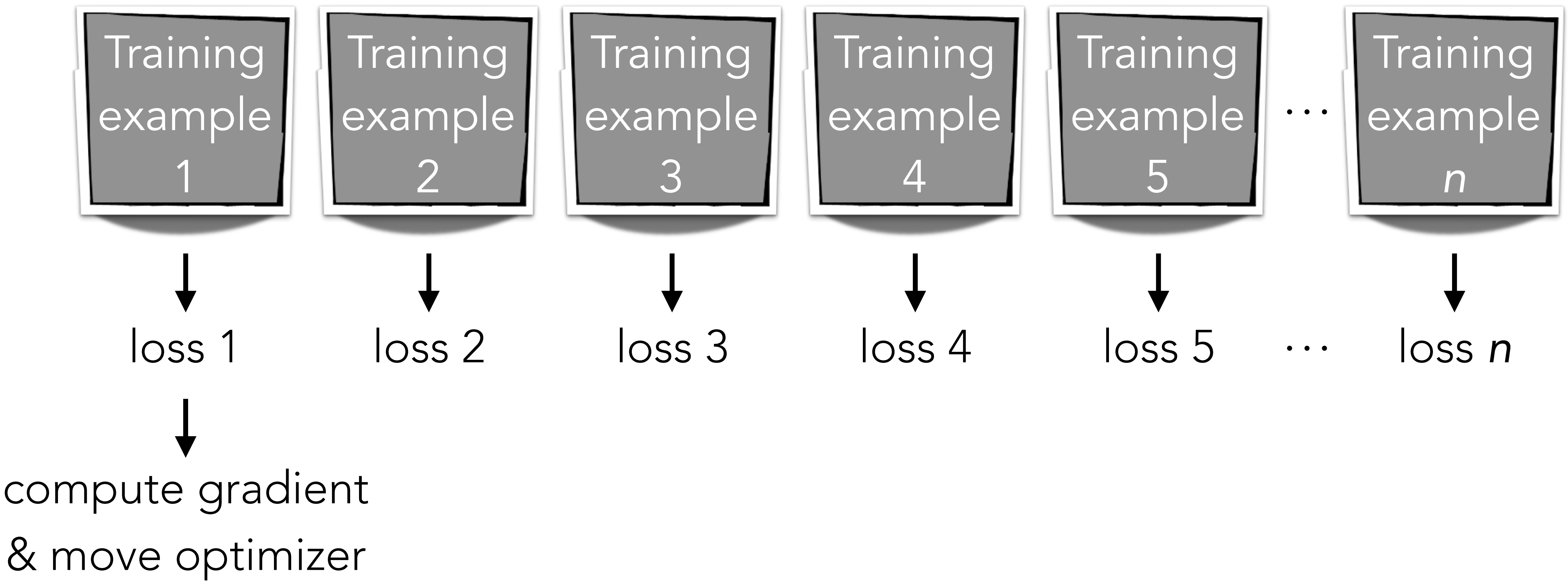
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



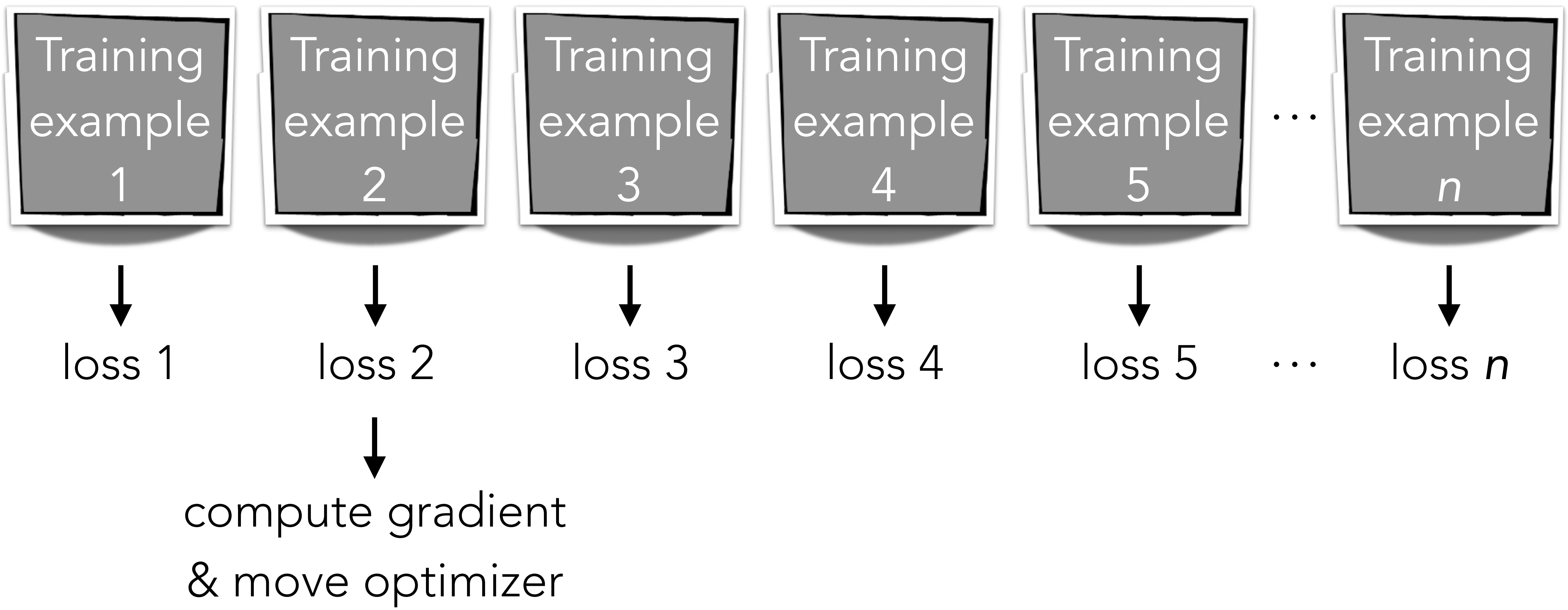
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the “full” gradient)

Stochastic Gradient Descent (SGD)



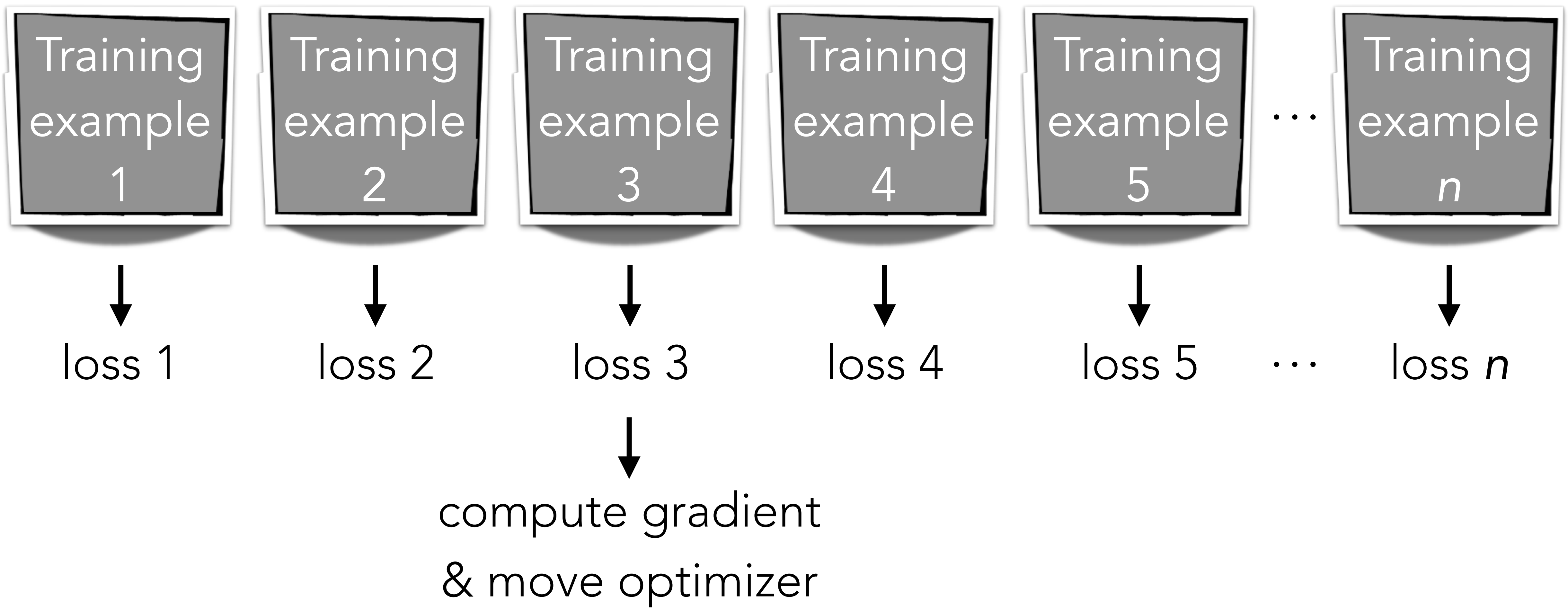
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



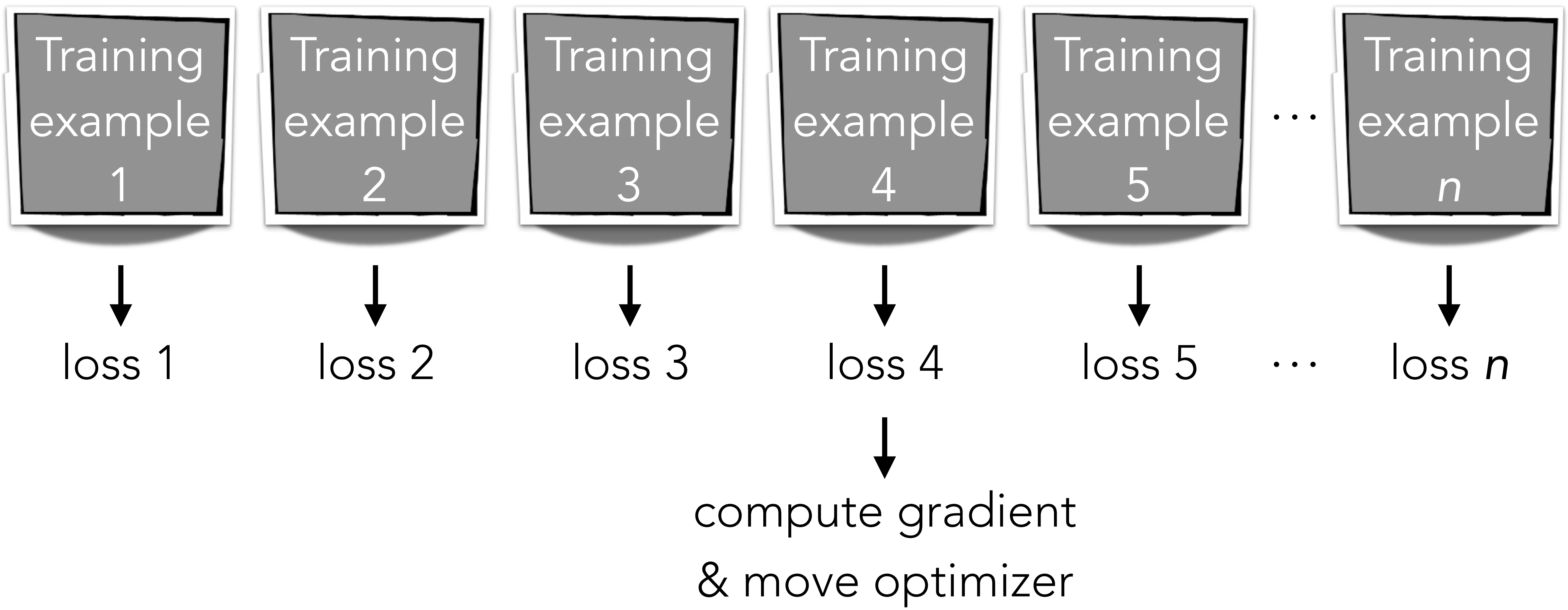
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



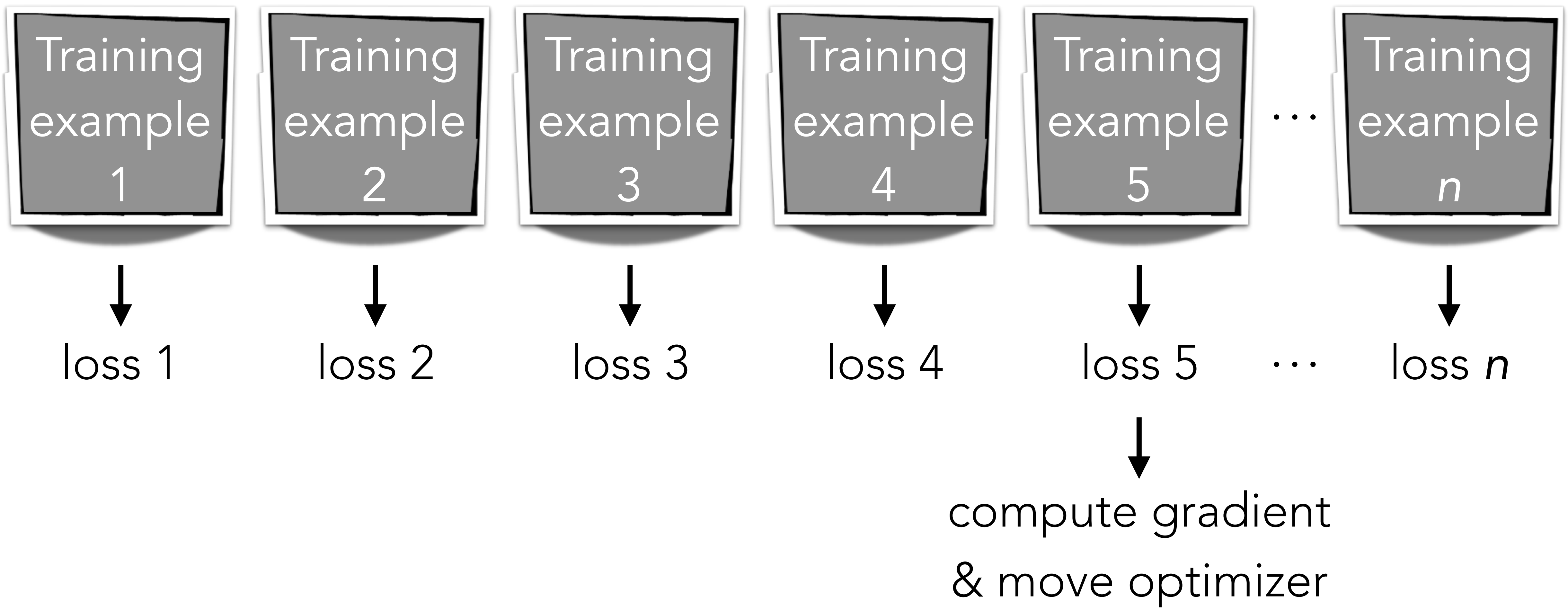
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the “full” gradient)

Stochastic Gradient Descent (SGD)



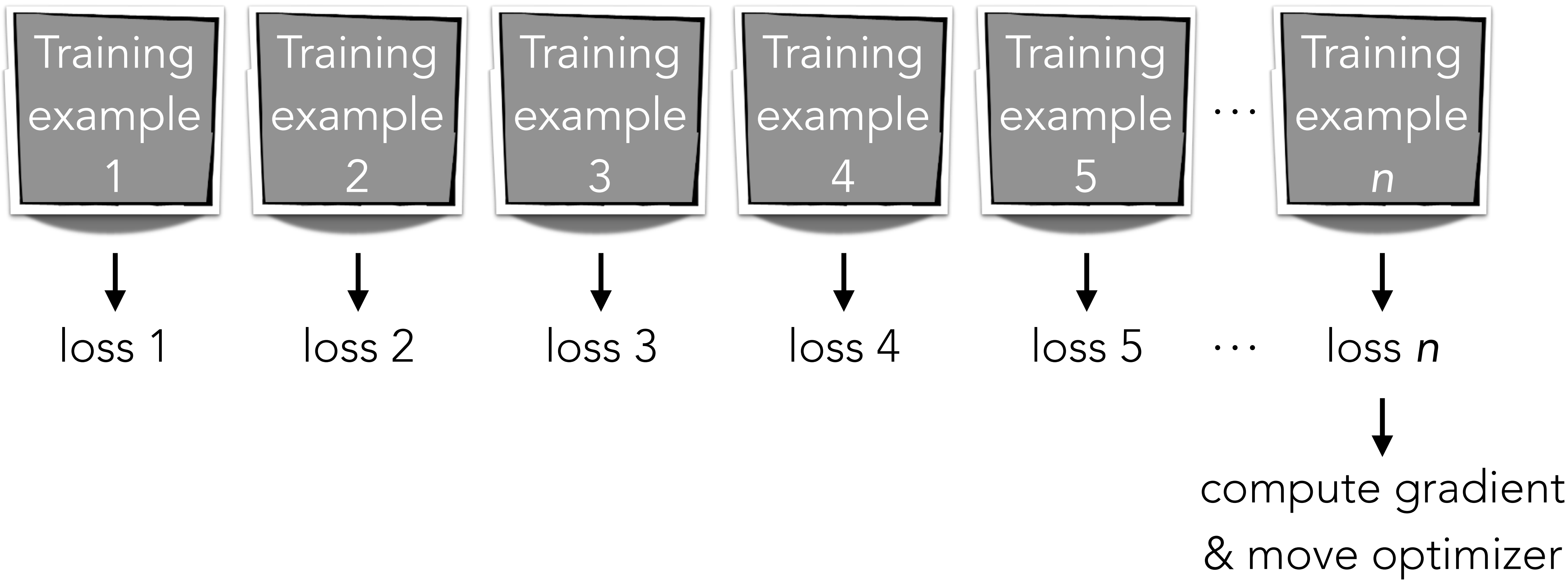
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



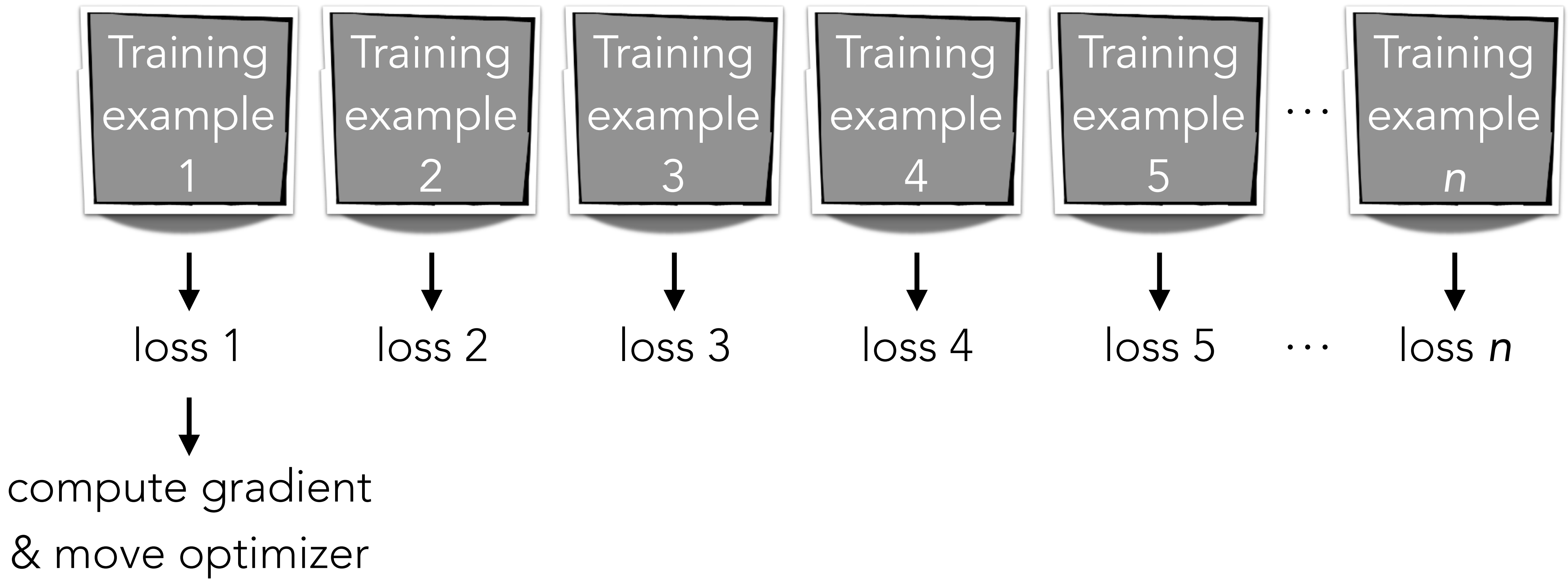
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



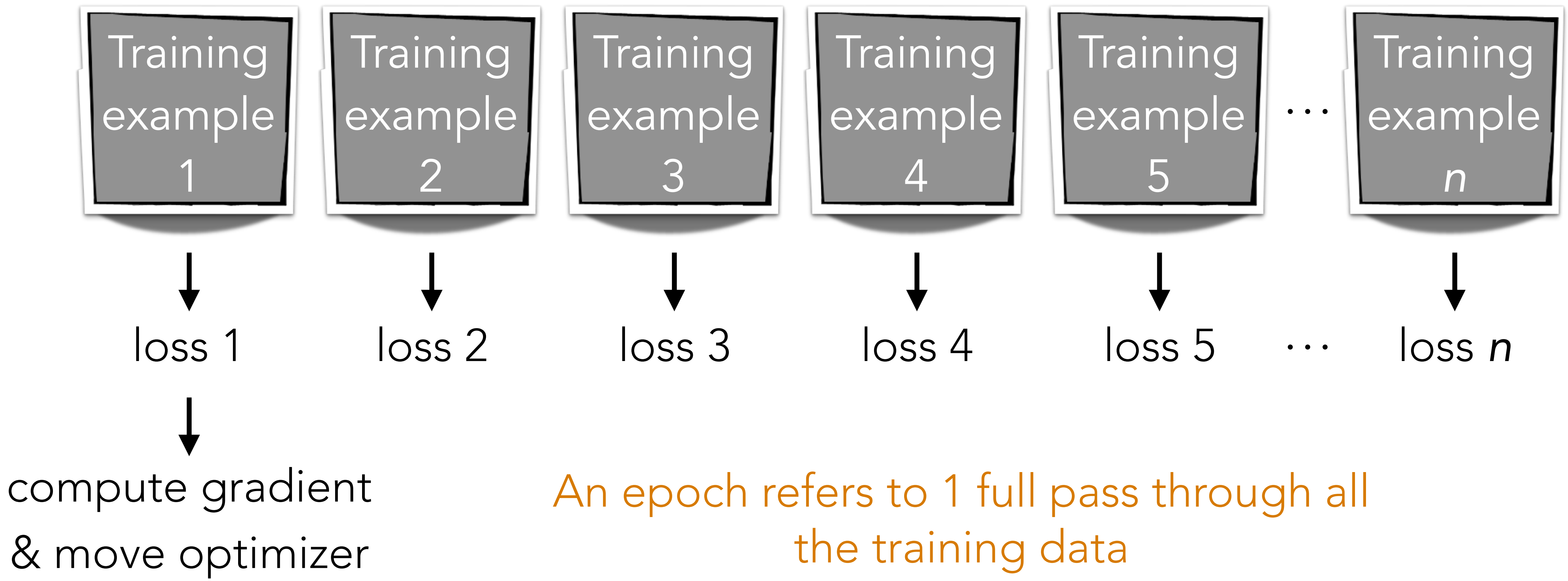
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

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)

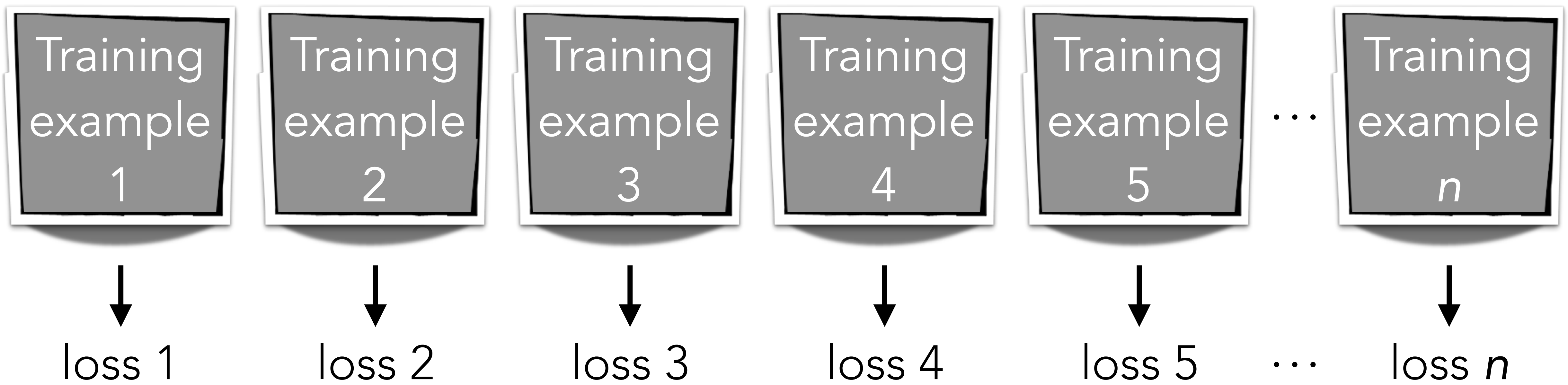
Stochastic Gradient Descent (SGD)



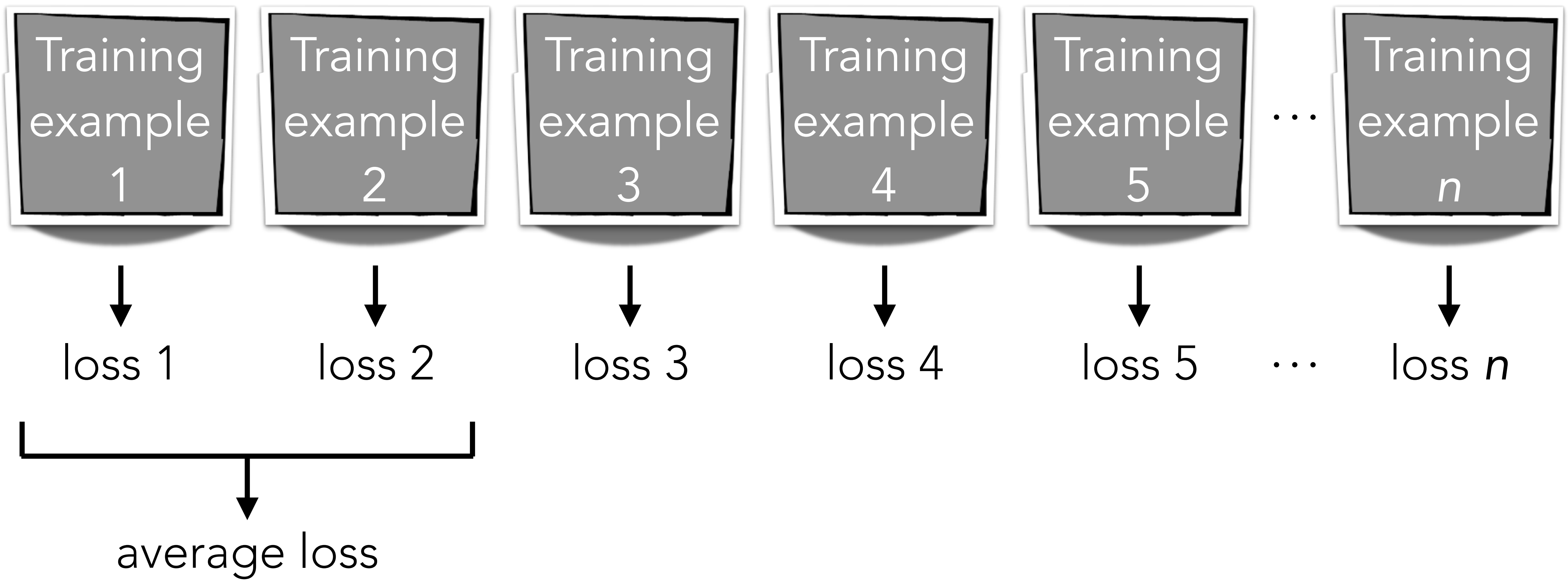
An epoch refers to 1 full pass through all the training data

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

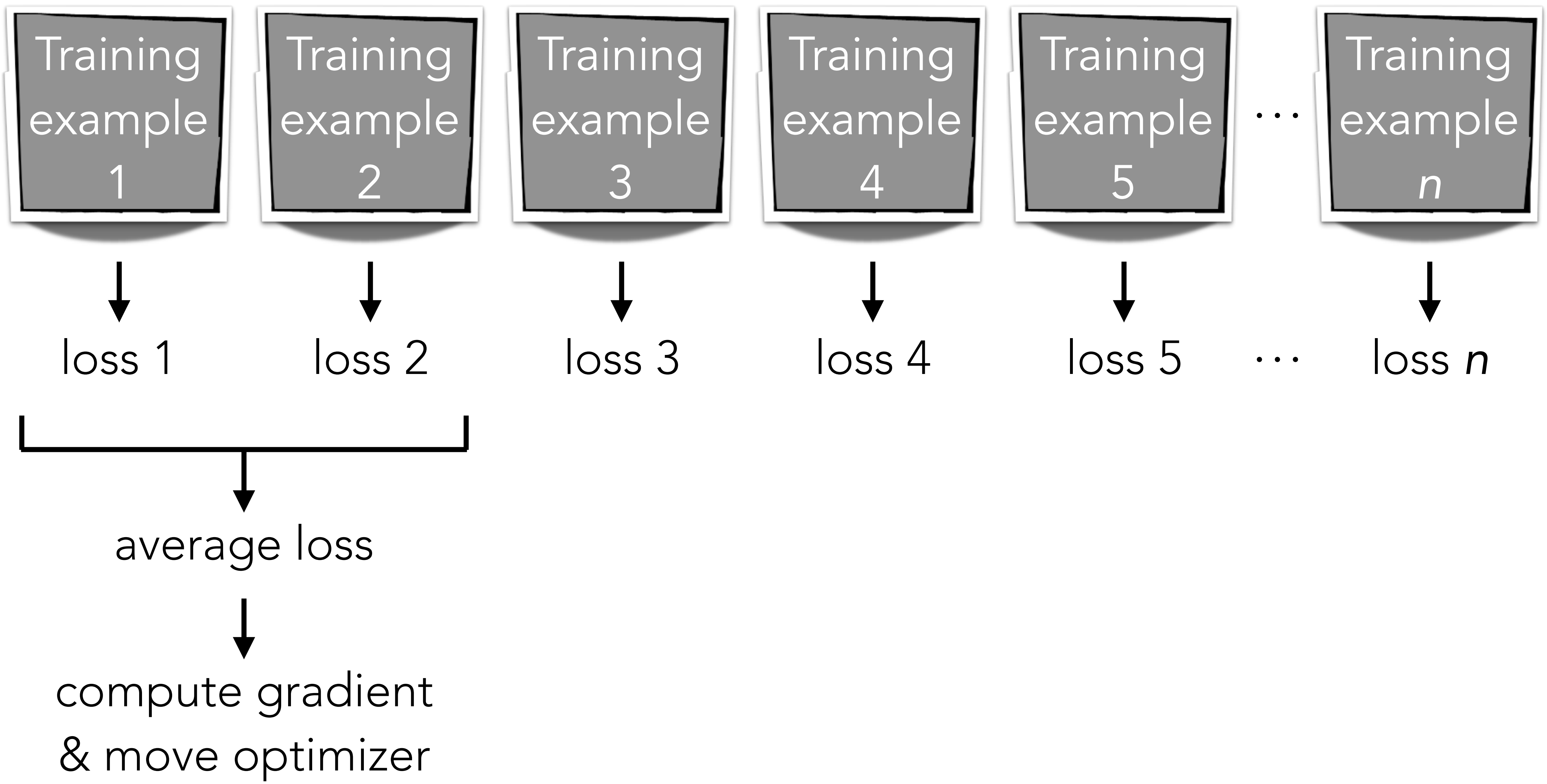
Minibatch Gradient Descent



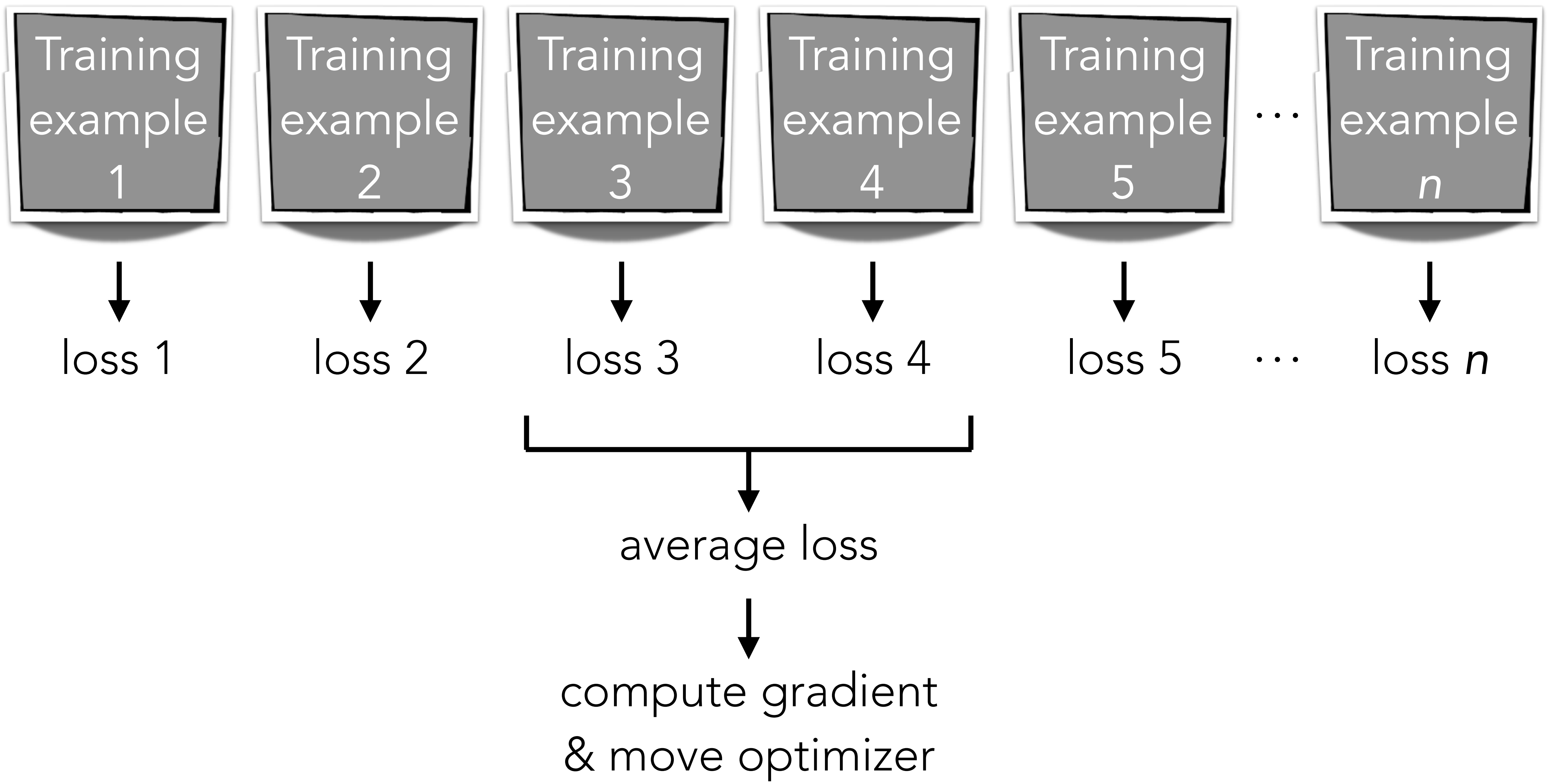
Minibatch Gradient Descent



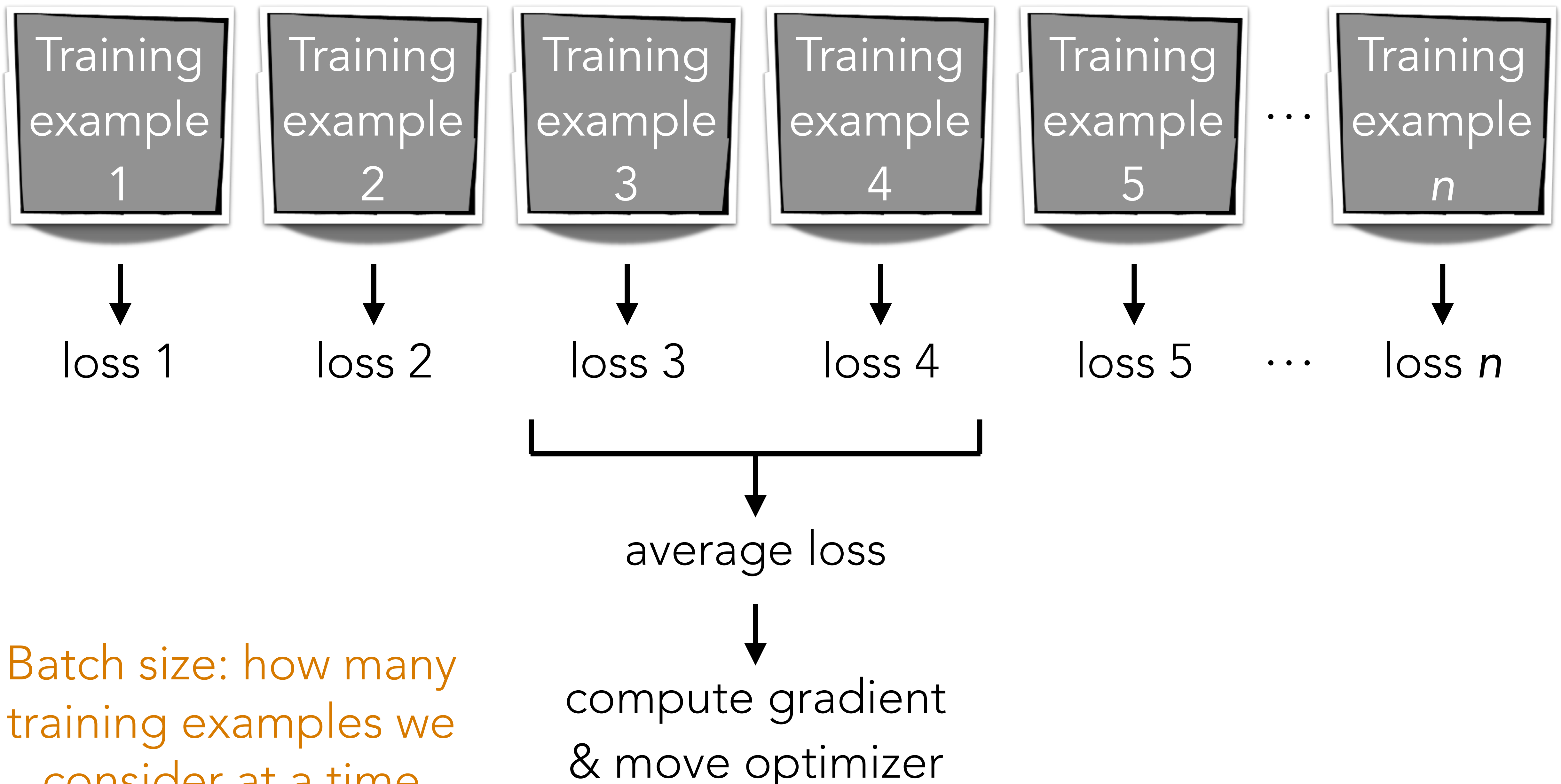
Minibatch Gradient Descent



Minibatch Gradient Descent



Minibatch Gradient Descent



Batch size: how many training examples we consider at a time (in this example: 2)

Best optimizer? Best learning rate? Best # of epochs? Best batch size?

Best optimizer? Best learning rate? Best # of epochs? Best batch size?

Active area of research

Best optimizer? Best learning rate? Best # of epochs? Best batch size?

Active area of research

Depends on problem, data, hardware, etc

Best optimizer? Best learning rate? 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!!!

UDA_pytorch_utils.py

A look at `UDA_pytorch_classifier_fit`, `UDA_pytorch_model_transform`, `UDA_pytorch_classifier_predict`

A special kind of RNN: an “LSTM”

(Flashback) Vanilla ReLU RNN

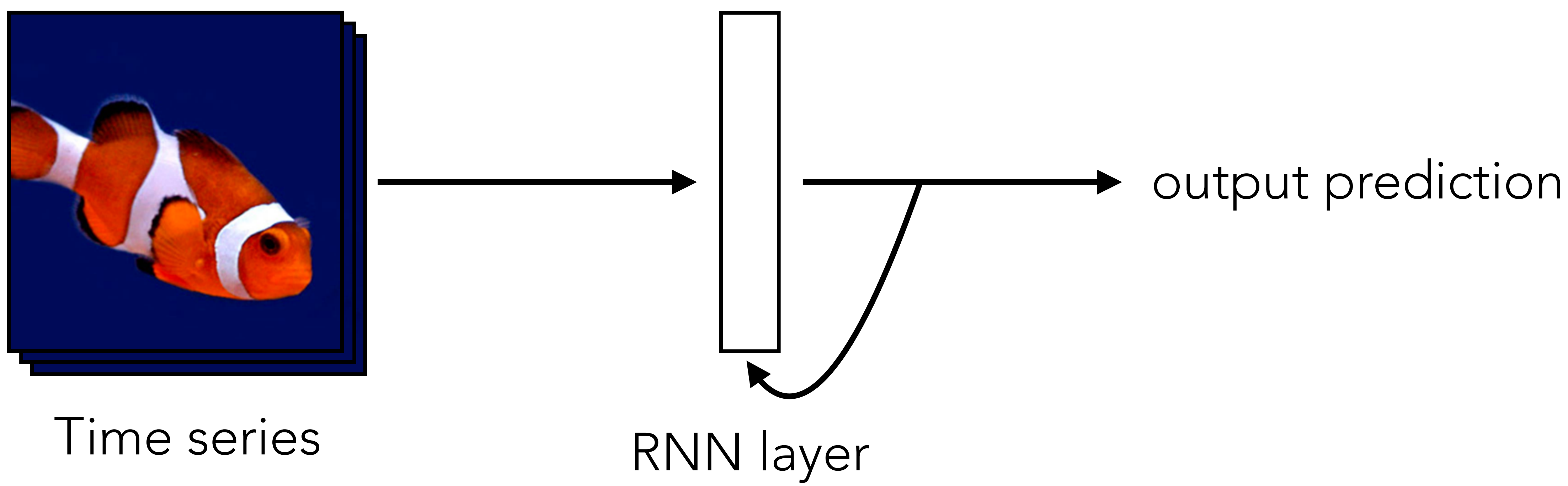
```
current_state = np.zeros(num_nodes)
outputs = []
for input in input_sequence:
  linear = np.dot(input, W.T) + b    \
          + np.dot(current_state, U.T)
  output = np.maximum(0, linear) # ReLU
  outputs.append(output)
  current_state = output
```

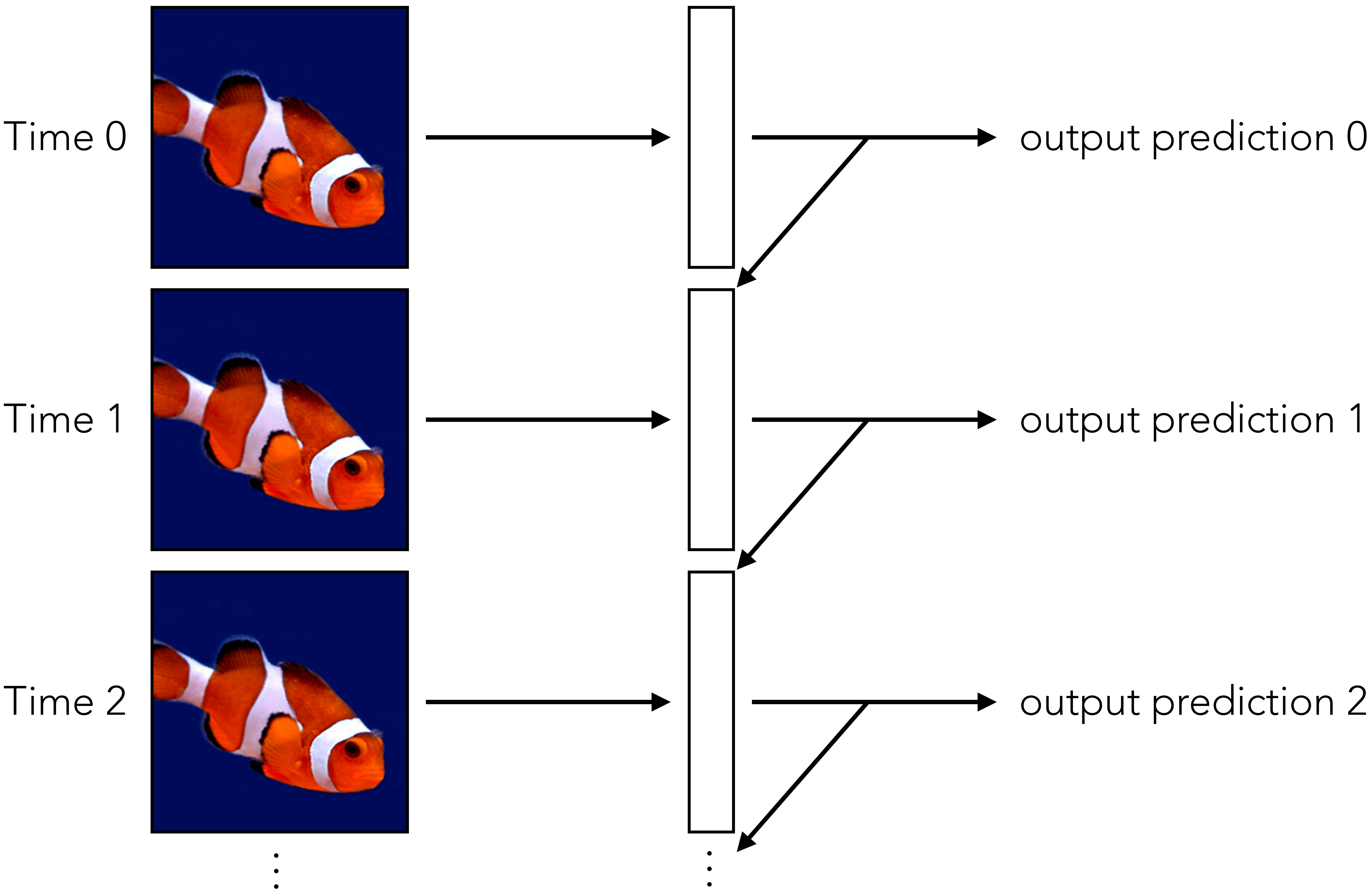
In general: there is an output at every time step

For simplicity, in today's lecture, we only use the very last time step's output

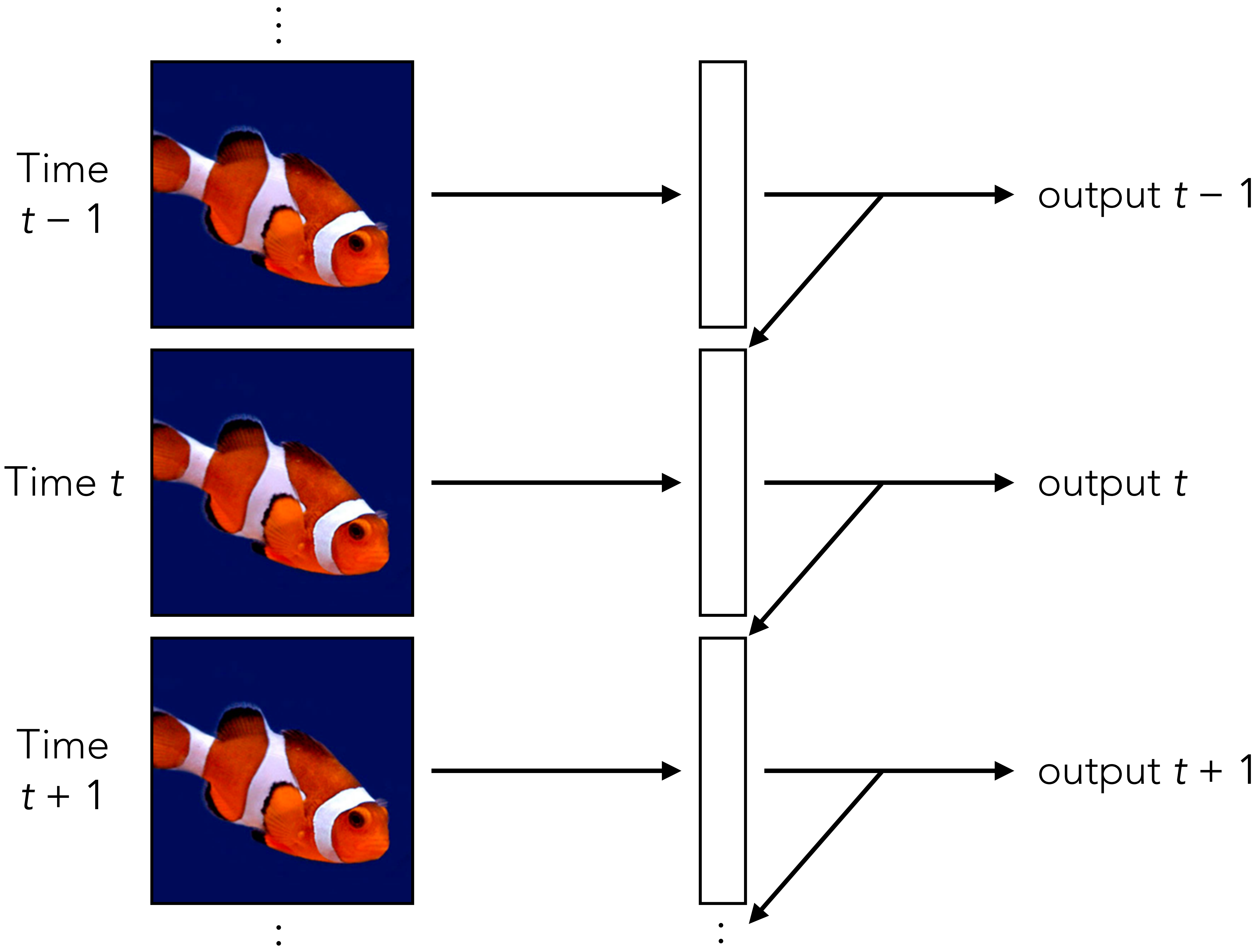
Vanilla ReLU RNN (another way to code it)

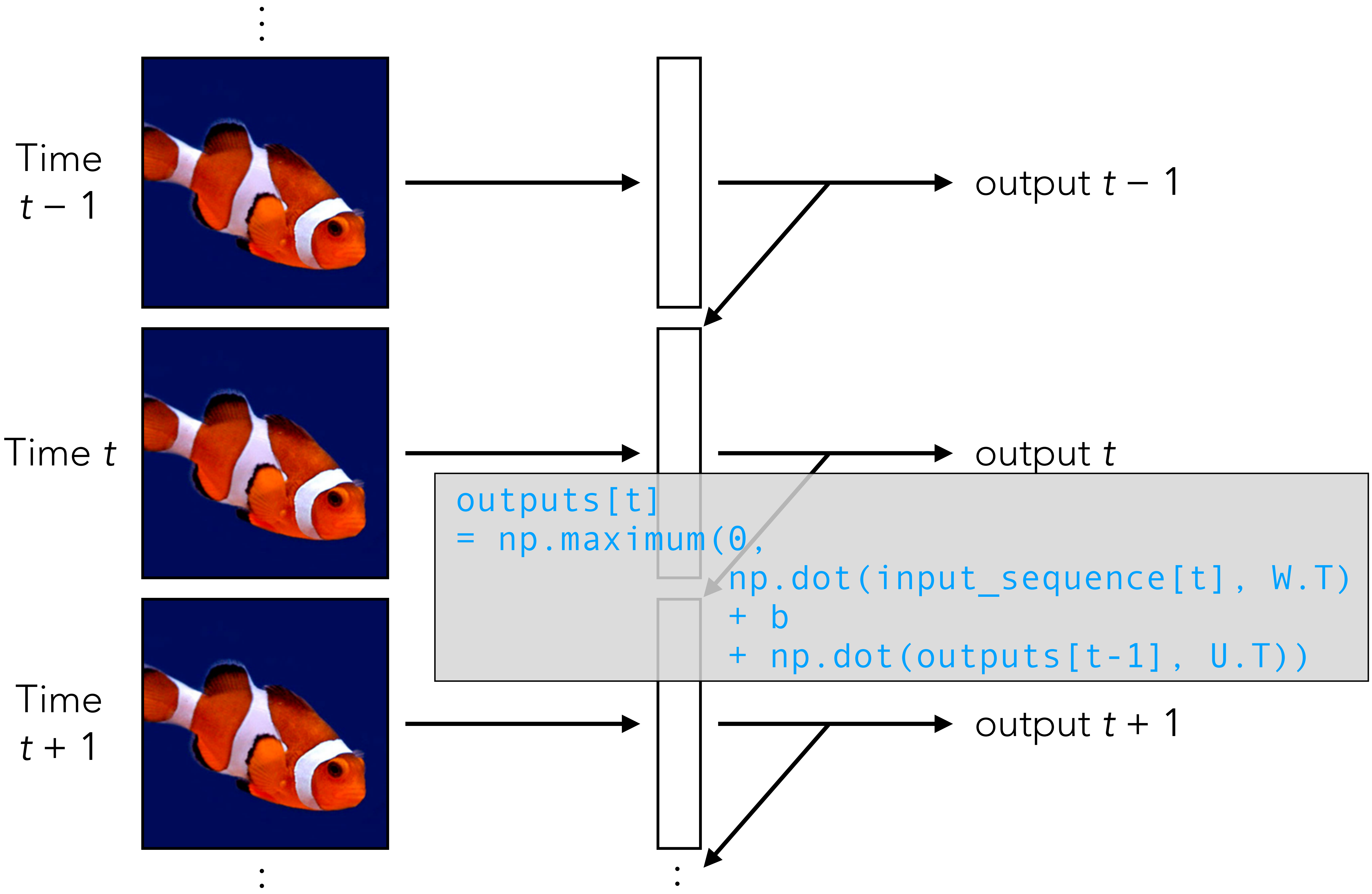
```
outputs = np.zeros((len(input_sequence), num_nodes))

for t in range(len(input_sequence)):

  if t == 0:

    outputs[t] = np.maximum(0,

        np.dot(input_sequence[t], W.T) + b)

  else:
    outputs[t] = np.maximum(

        0,

        np.dot(input_sequence[t], W.T) + b

        + np.dot(outputs[t-1], U.T)

    )
```

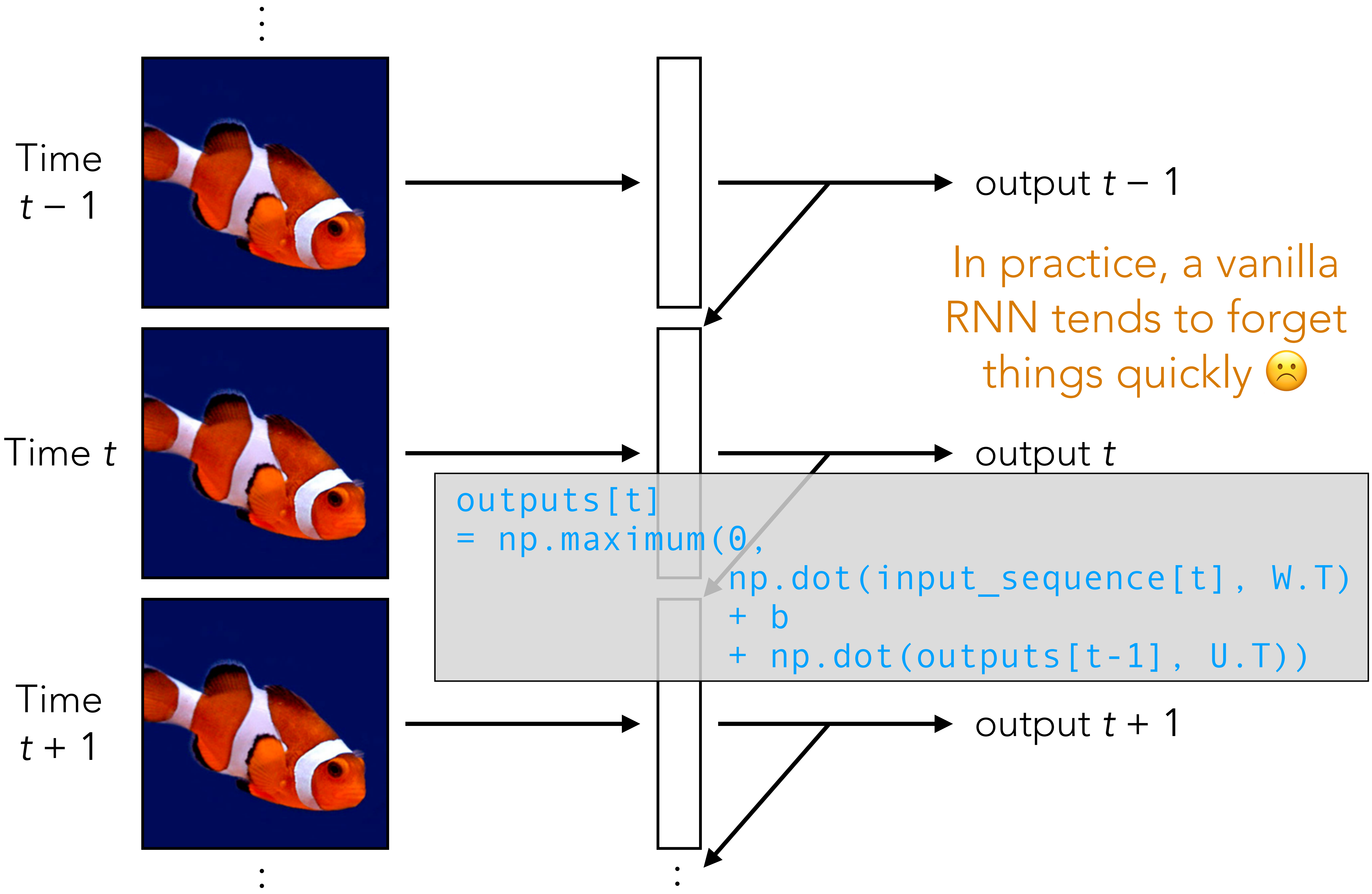

Time series
RNN layer
output prediction



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

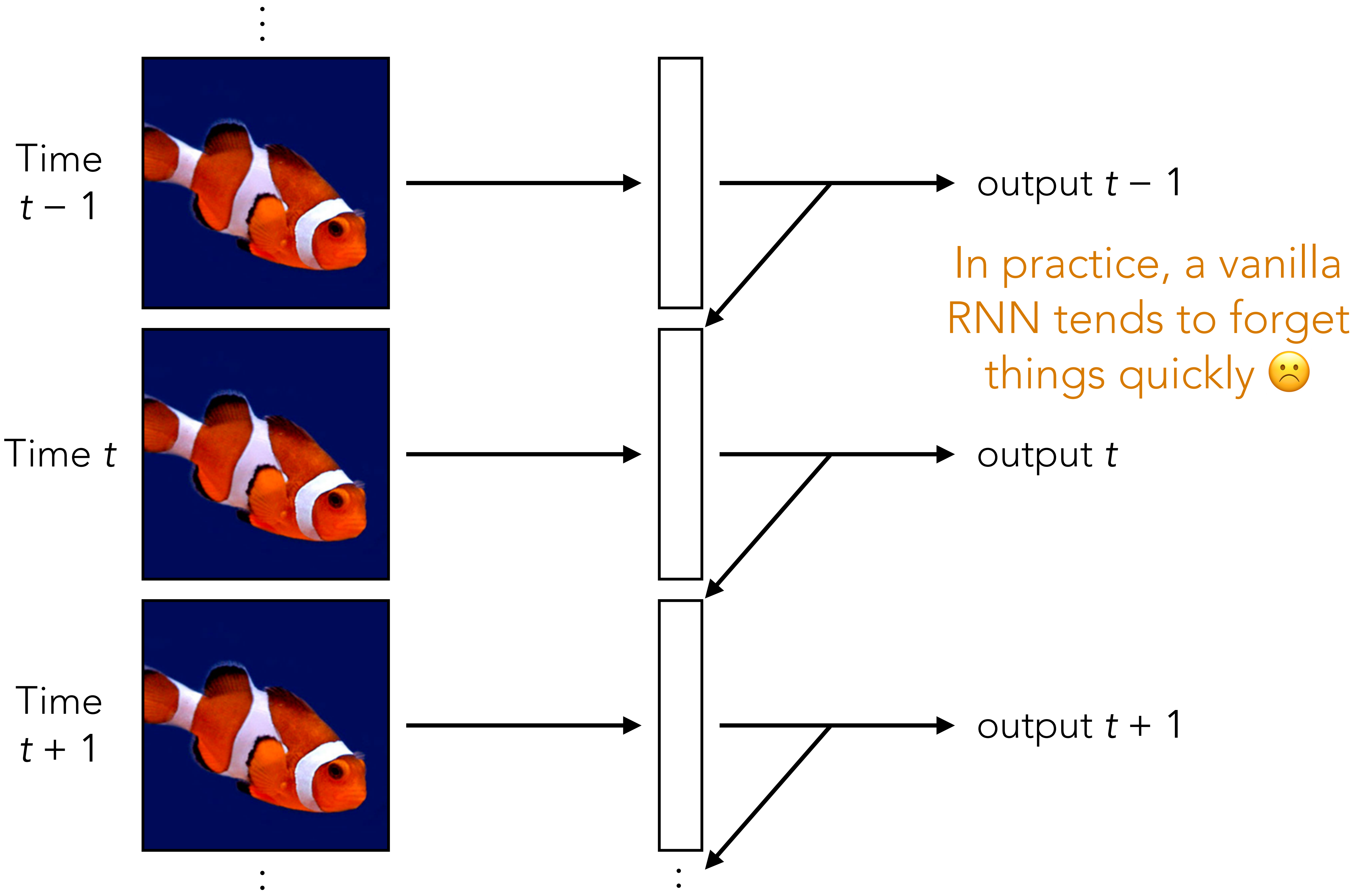

Time
t − 1
Time t
Time
t + 1
output t − 1
output t
output t + 1


Time
t − 1
Time t
Time
t + 1
output t − 1
output t
output t + 1
outputs[t]
= np.maximum(0,
np.dot(input_sequence[t], W.T)
+ b
+ np.dot(outputs[t-1], U.T))

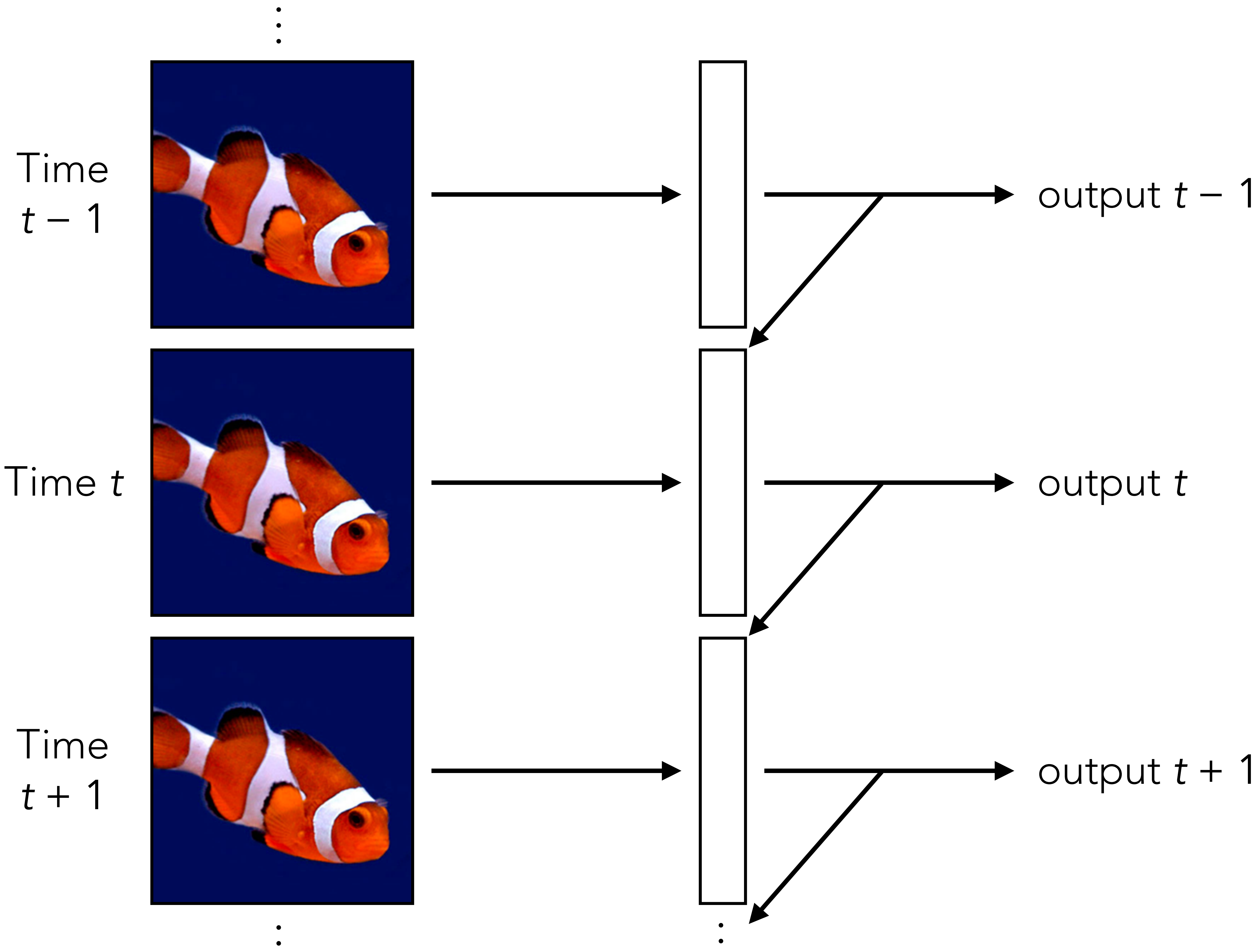


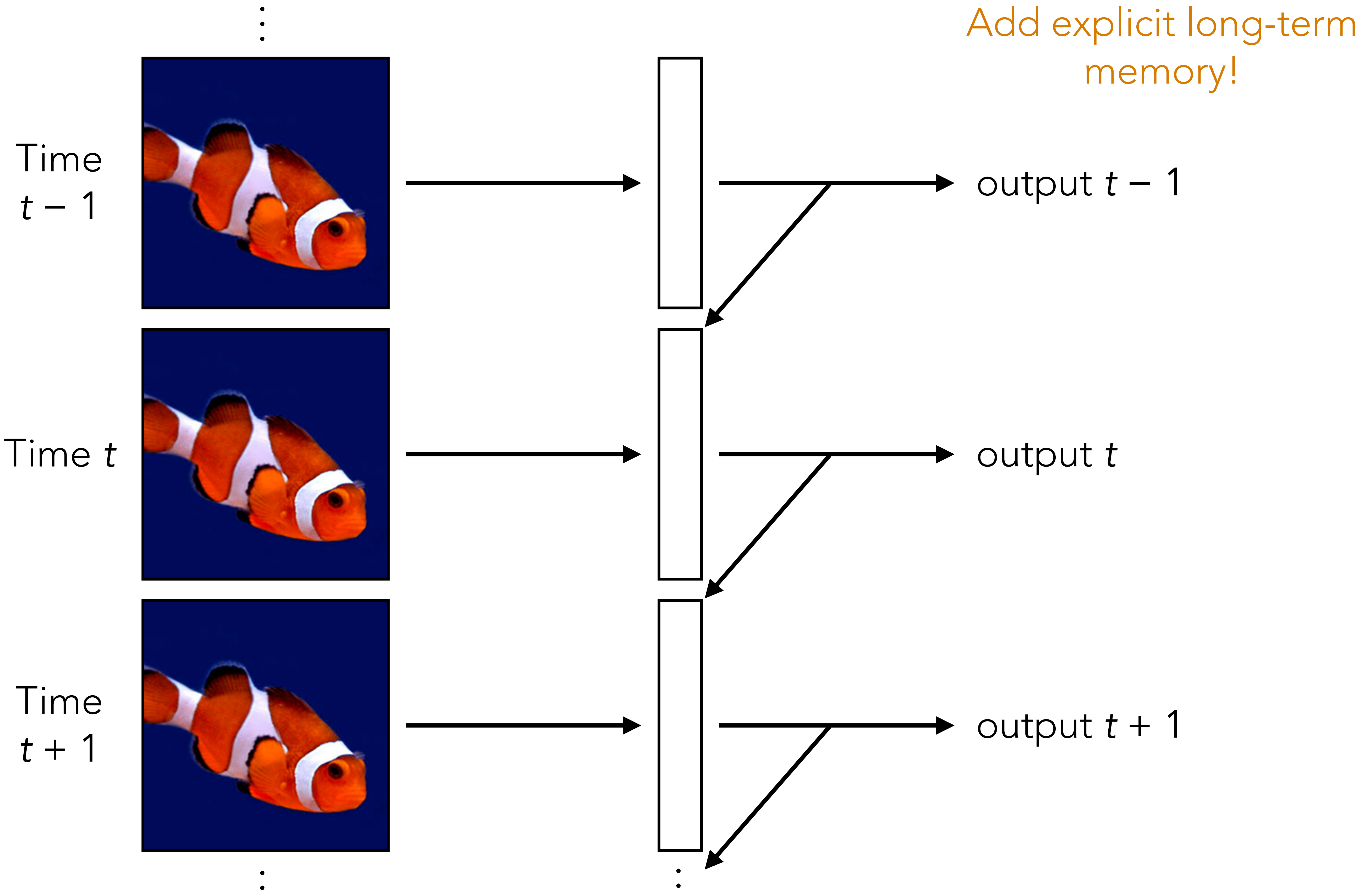
In practice, a vanilla RNN tends to forget things quickly ☹️

```
outputs[t]
= np.maximum(0,
             np.dot(input_sequence[t], W.T)
             + b
             + np.dot(outputs[t-1], U.T))
```



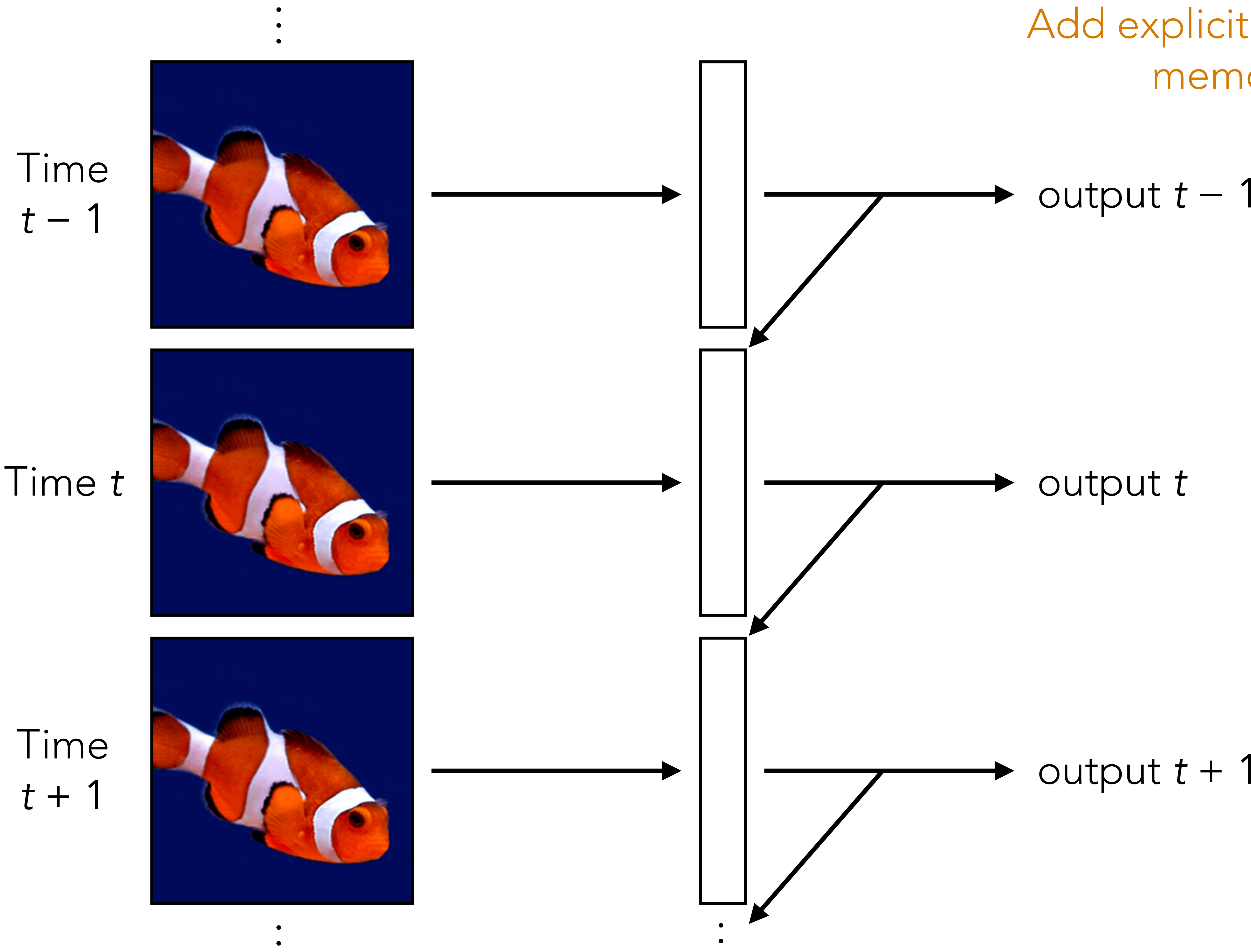
Time
$t - 1$
output $t - 1$
In practice, a vanilla RNN tends to forget things quickly ☹️
Time t
output t
Time
$t + 1$
output $t + 1$

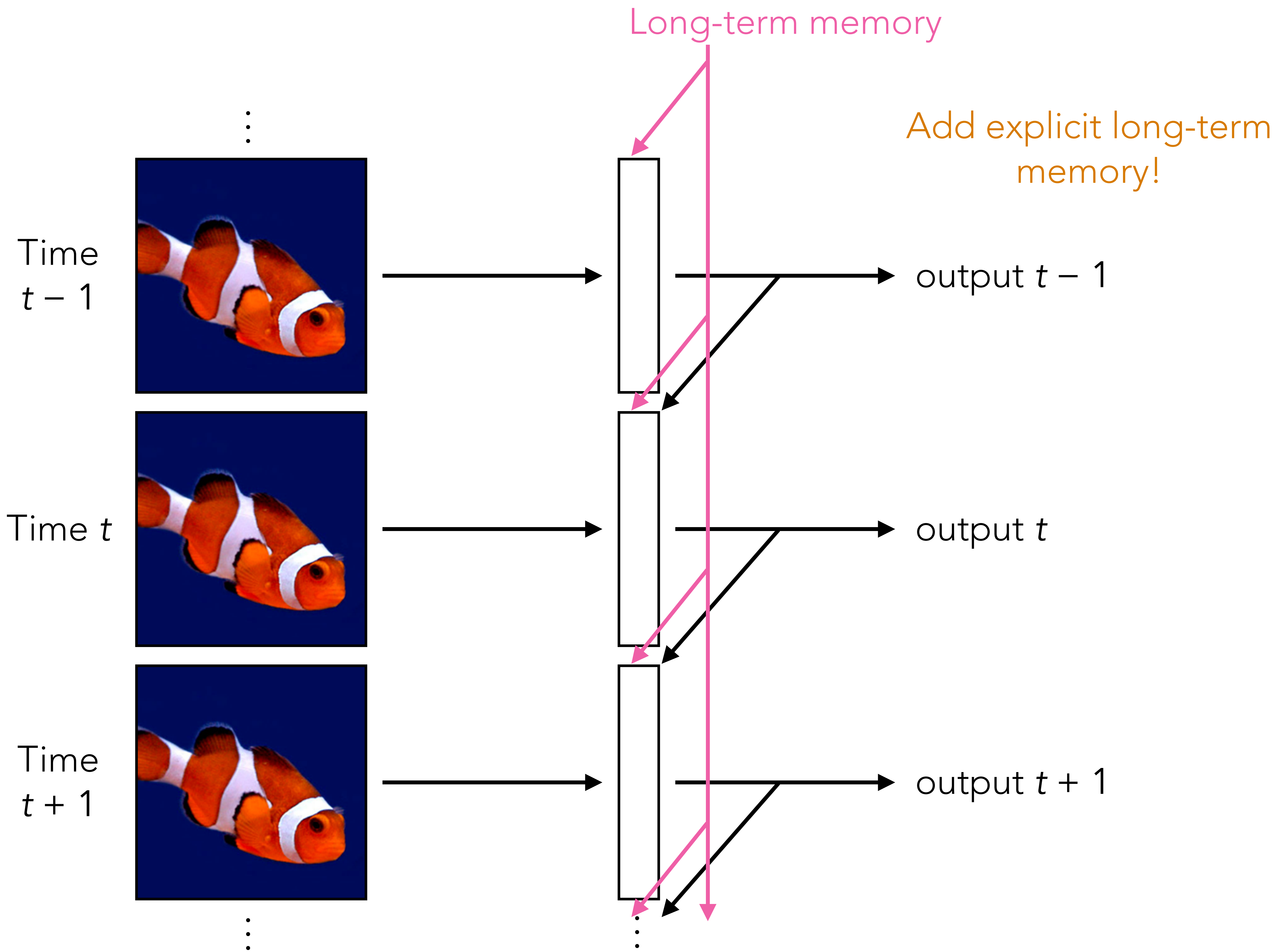

Time
t − 1
output t − 1
Time t
output t
Time
t + 1
output t + 1

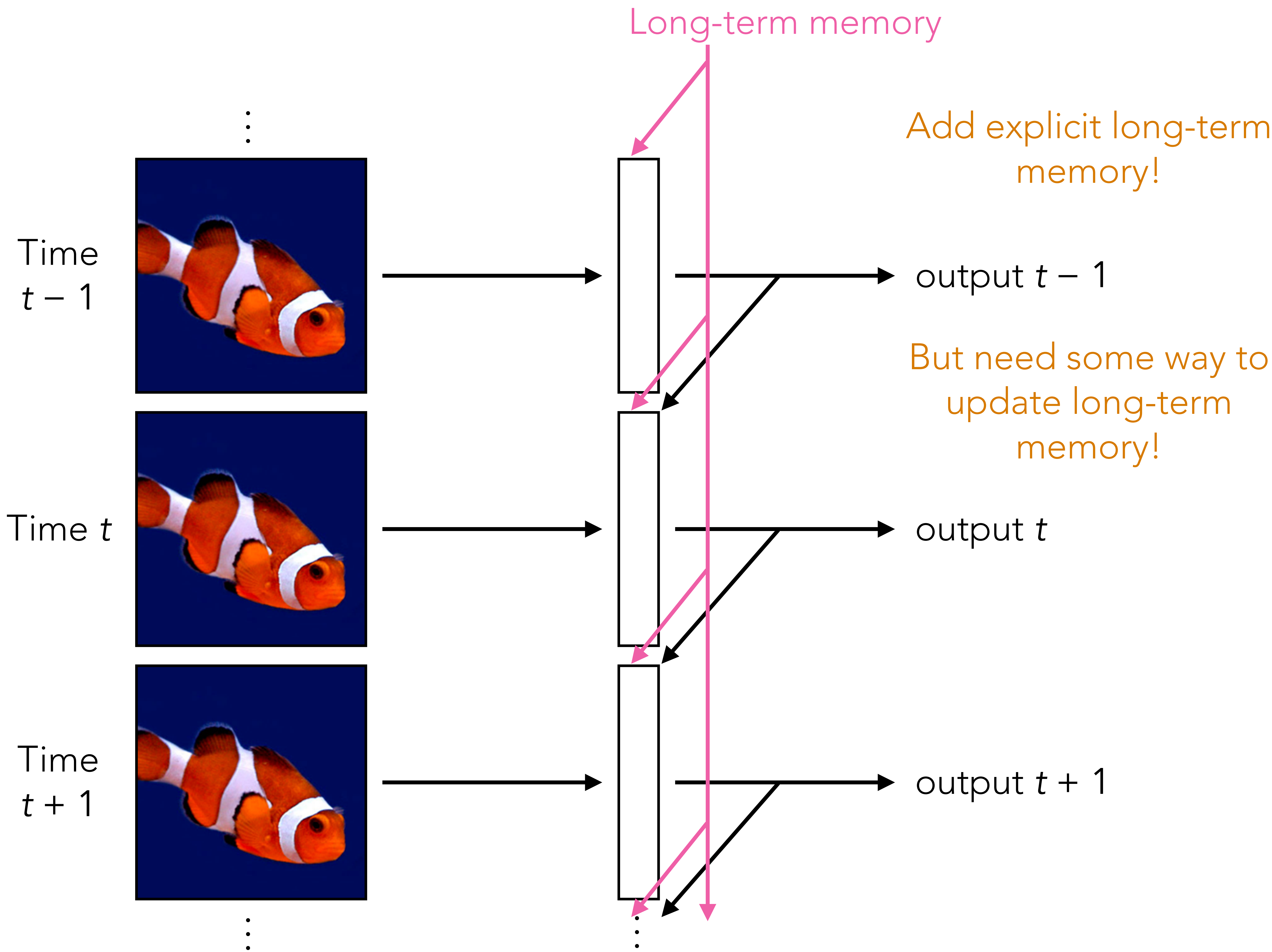

Time
$t-1$
Time t
Time
$t+1$
output $t-1$
output t
output $t+1$
Add explicit long-term
memory!

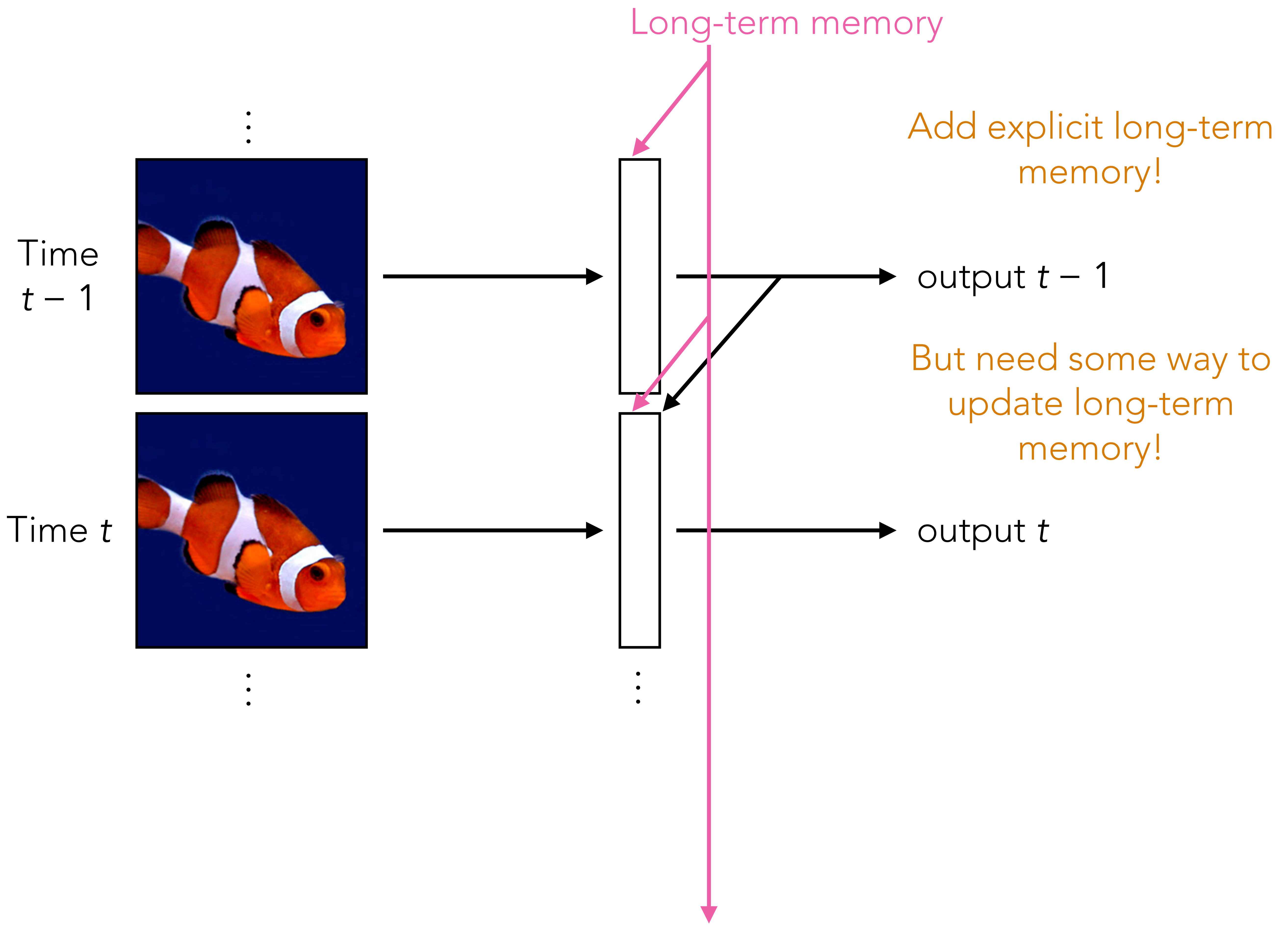
Long-term memory

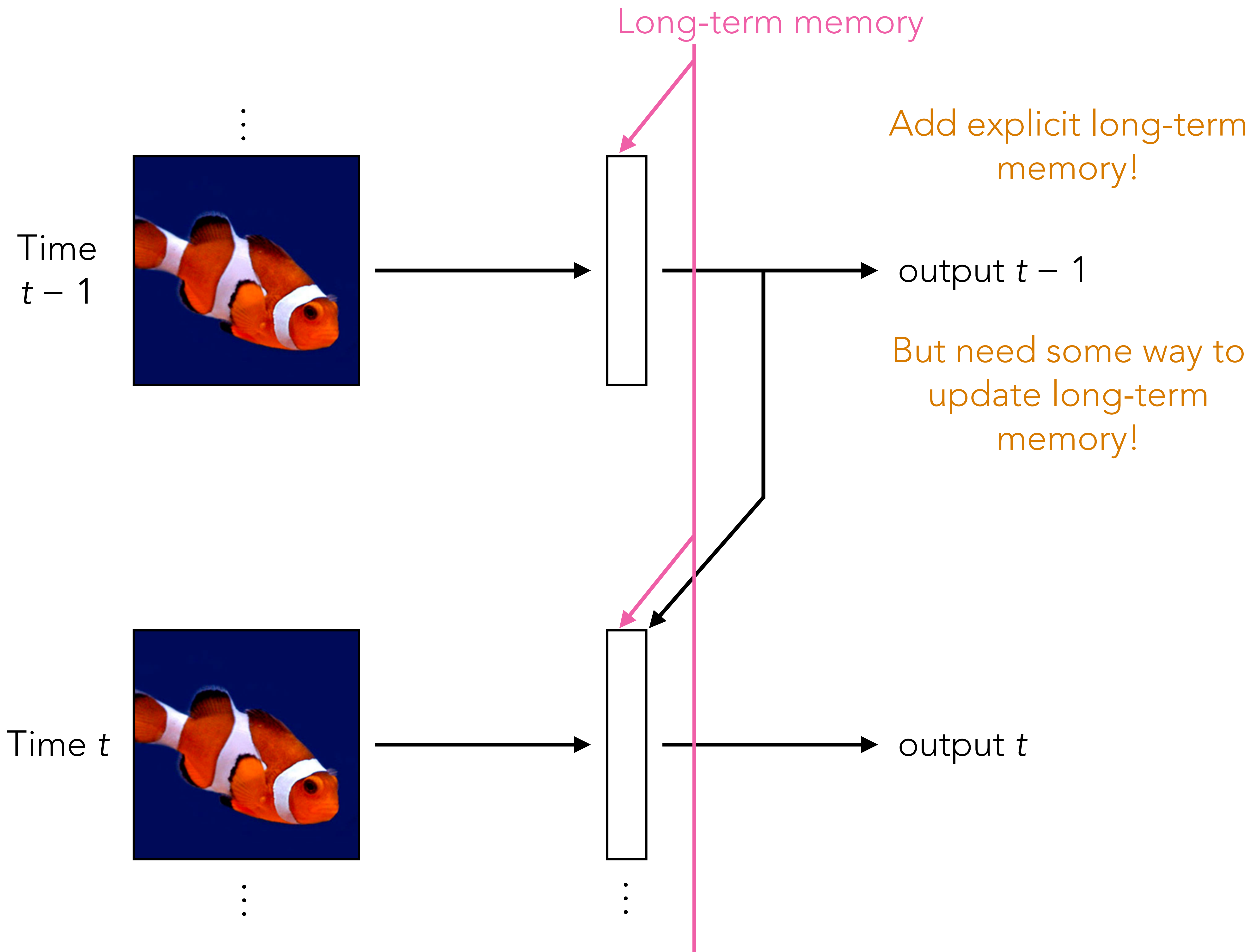
Add explicit long-term memory!



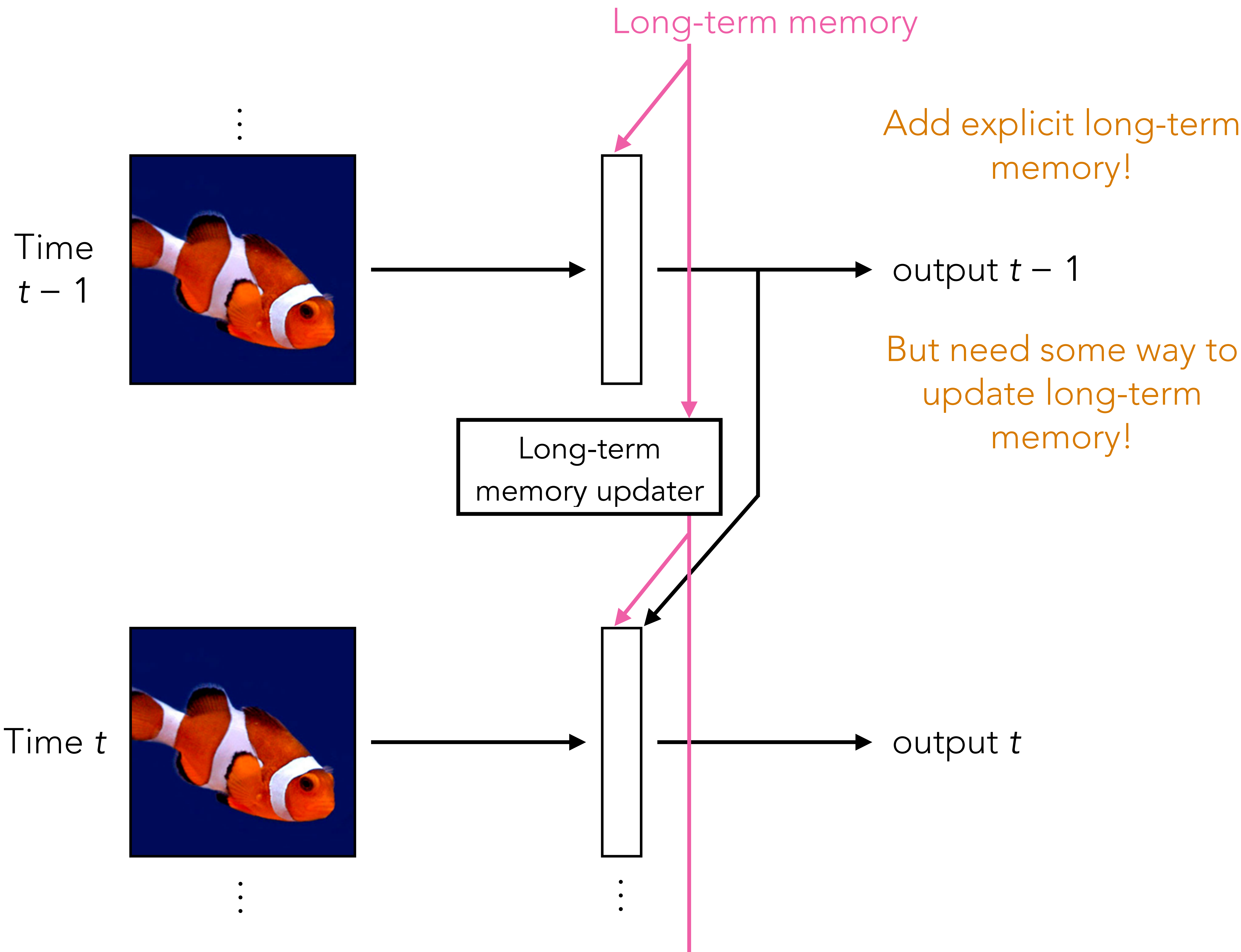

Long-term memory
Add explicit long-term memory!
Time t − 1
Time t
Time t + 1
output t − 1
output t
output t + 1


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
Time t + 1
output t + 1

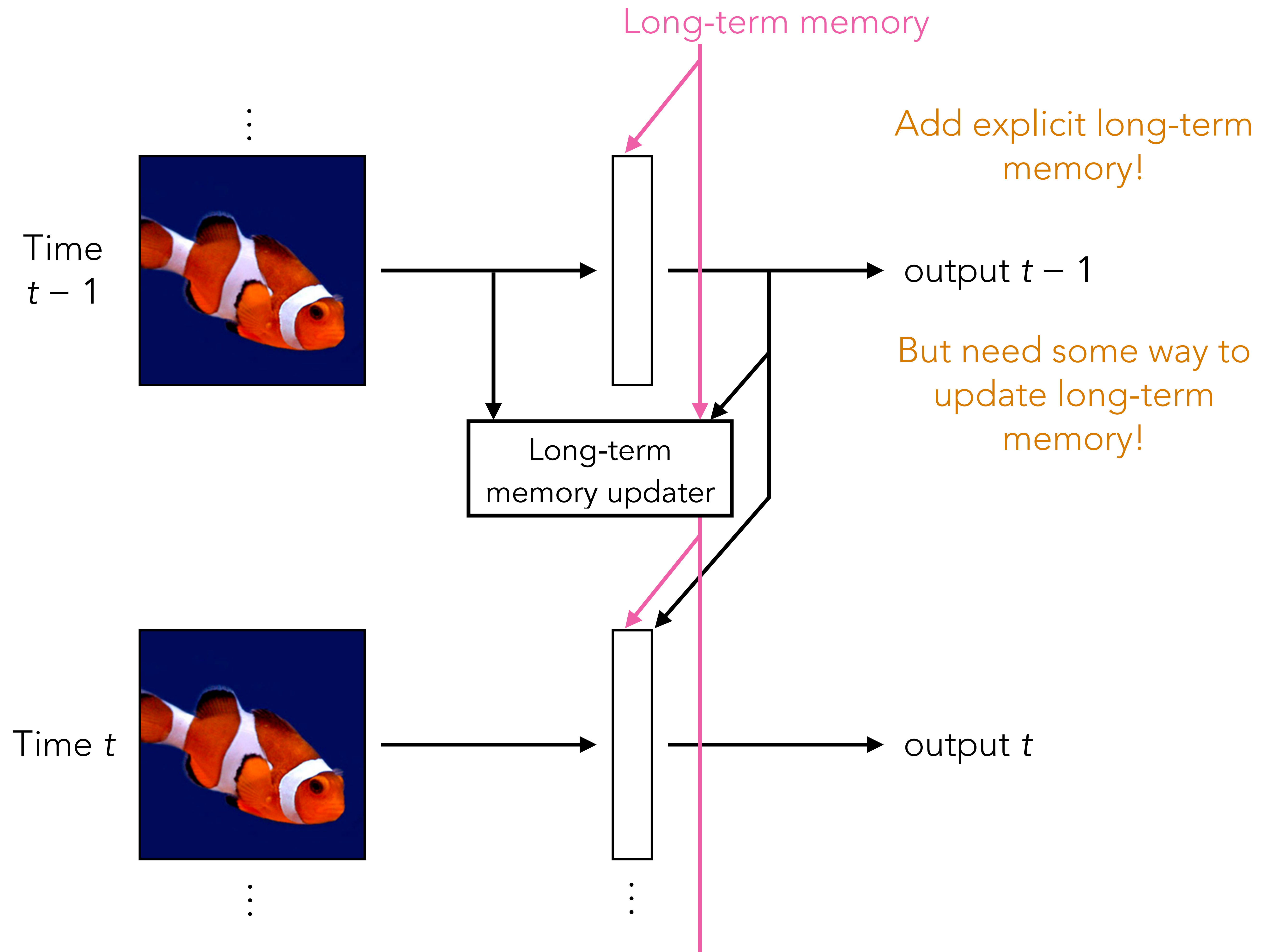

Long-term memory
Add explicit long-term memory!
Time
t − 1
output t − 1
But need some way to update long-term memory!
Time t
output t



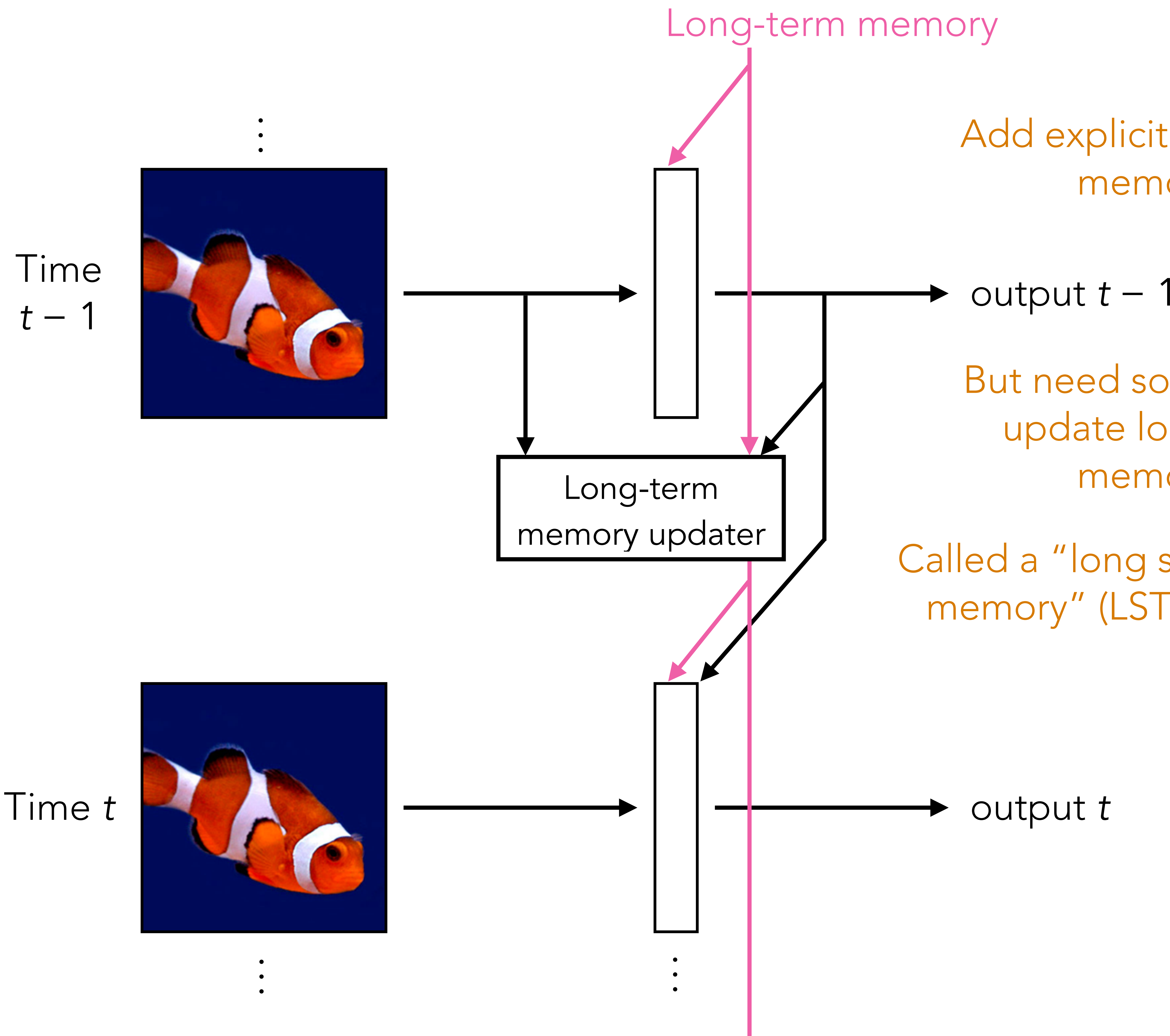
Long-term memory
Add explicit long-term memory!
Time t − 1
output t − 1
But need some way to update long-term memory!
Time t
output t



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



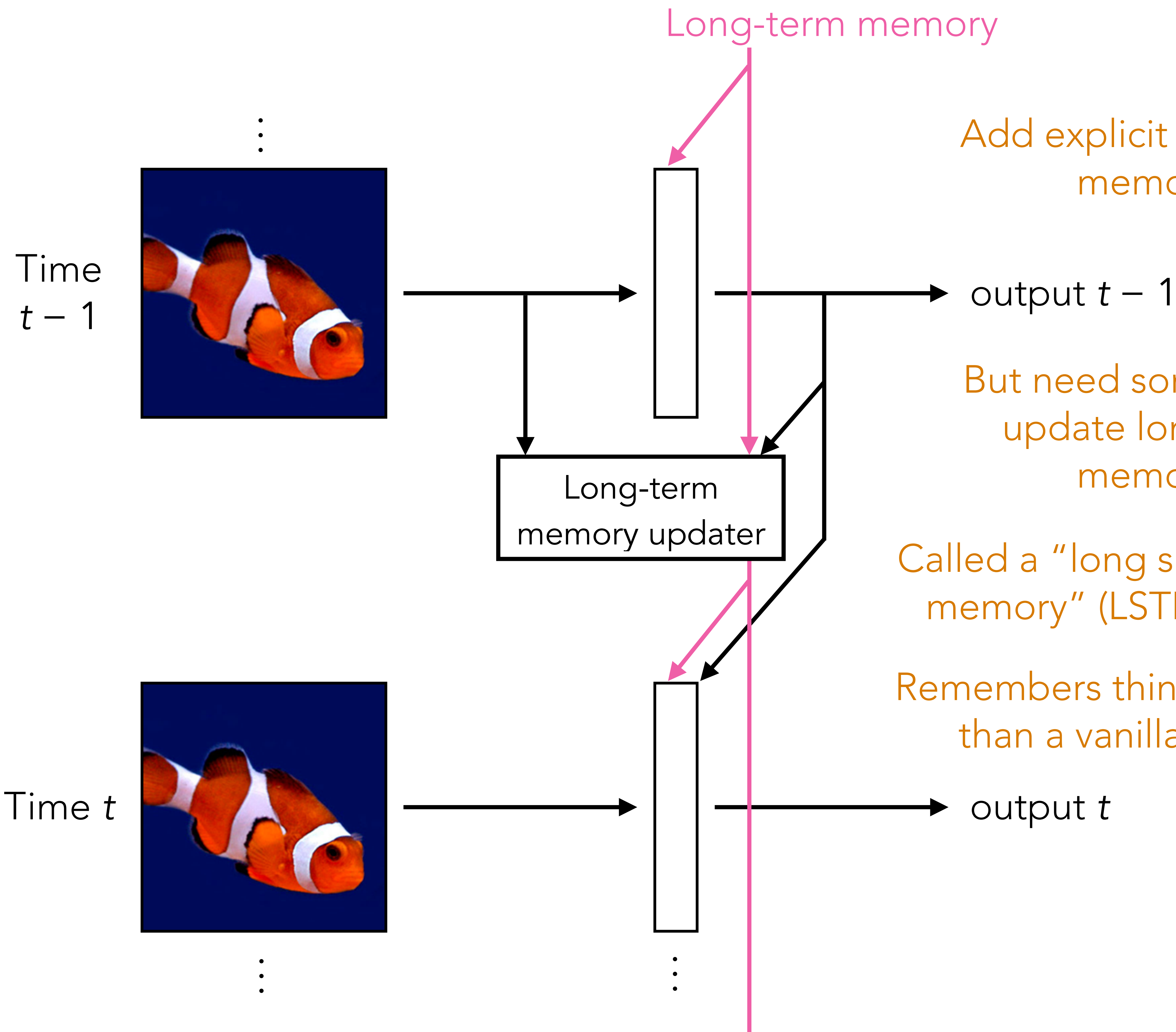
Long-term memory
Add explicit long-term memory!
Time t − 1
output t − 1
But need some way to update long-term memory!
Long-term memory updater
Time t
output t



Add explicit long-term memory!

But need some way to update long-term memory!

Called a "long short-term memory" (LSTM) RNN



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 a vanilla RNN

Warning: PyTorch’s implementation of a vanilla RNN is different from the one in lecture due to a technicality

pytorch / torch / nn / modules / **rnn.py**

Code Blame 1825 lines (1604 loc) · 72.5 KB ·

```
    __constants__ = [
                b_ih = Parameter(torch.empty(gate_size, **factory_kwargs))
                # Second bias vector included for CuDNN compatibility. Only one
                # bias vector is needed in standard definition.
                b_hh = Parameter(torch.empty(gate_size, **factory_kwargs))
```

In particular, PyTorch’s `RNN` class uses an extra bias vector that is *not* actually standard…

Analyzing Times Series with CNNs

Analyzing Times Series with CNNs

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

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!)

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*

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 data do not have long-range dependencies that require long-term memory, CNNs can do well already!

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 data do not have long-range dependencies that require long-term memory, CNNs can do well already!

  ⇒ If you need long-term memory or time series with different lengths, use RNNs (not the vanilla one) or transformers