Carnegie Mellon University
Heinz College

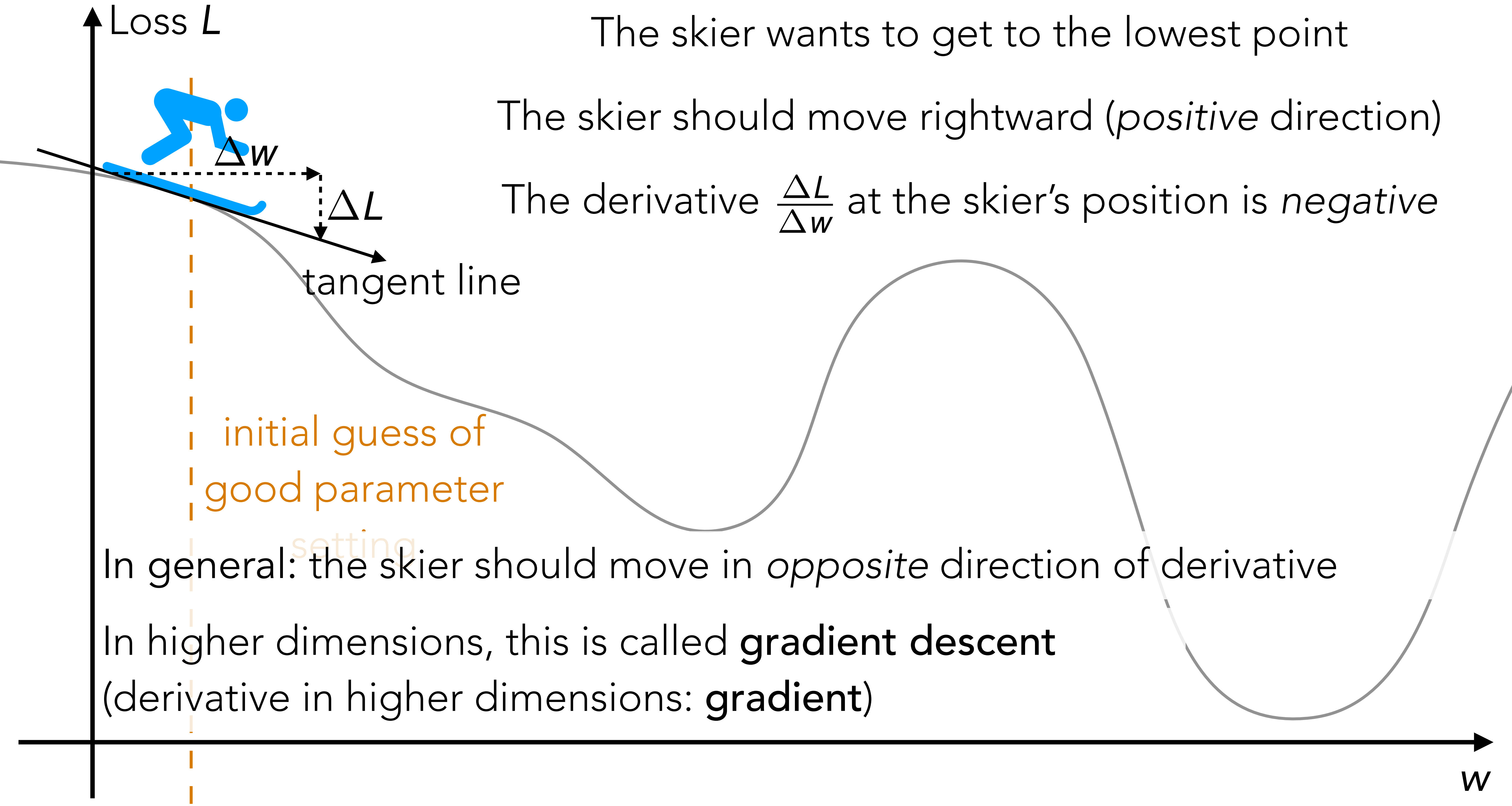
95-865 Unstructured Data Analytics

Recitation: More on minibatch gradient descent, RNNs, and transformers

Slides by George H. Chen & Shahriar Noroozizadeh

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)

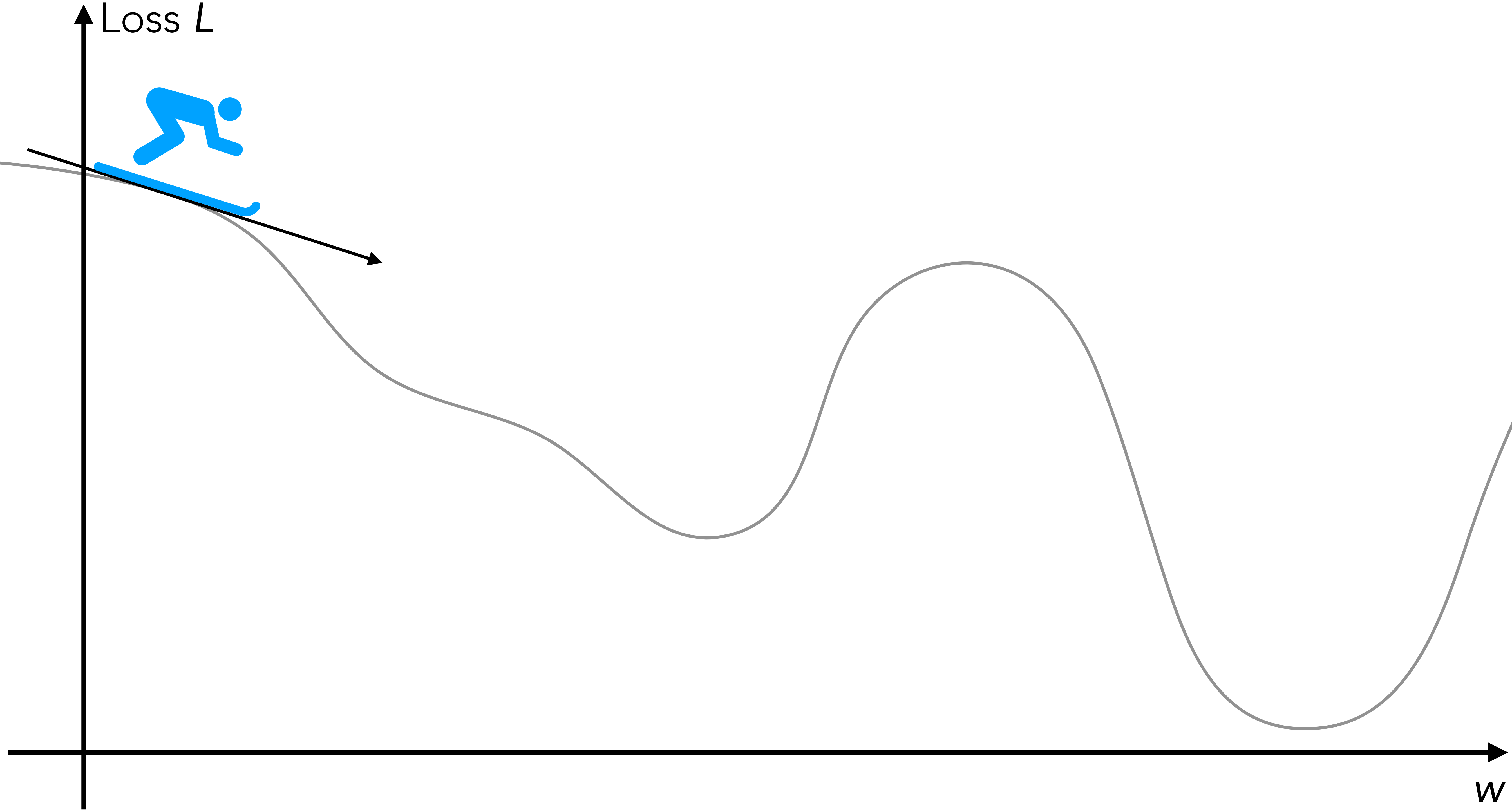
The derivative $\frac{\Delta L}{\Delta w}$ at the skier's position is *negative*

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

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

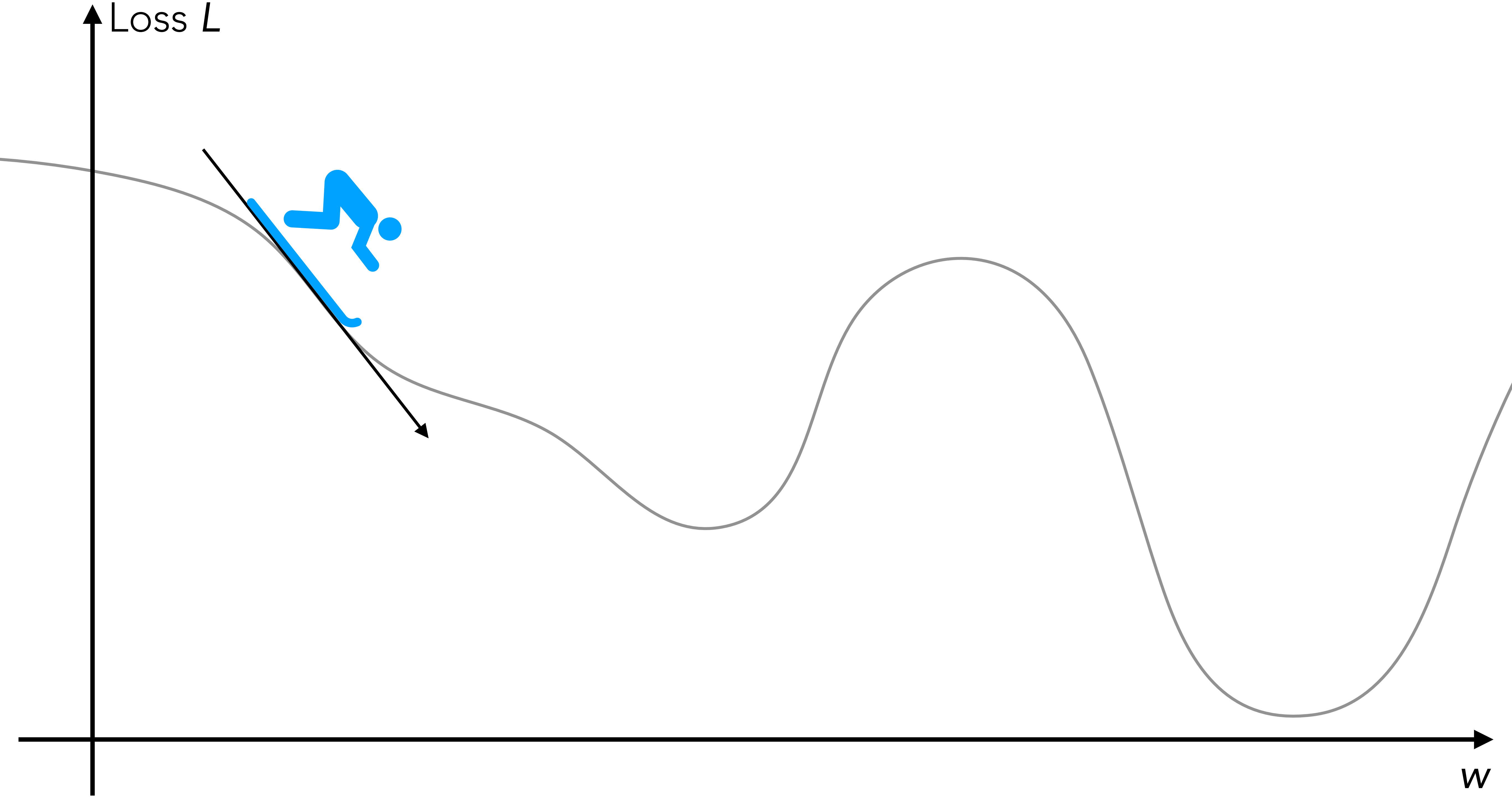
Learning a Deep Net

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



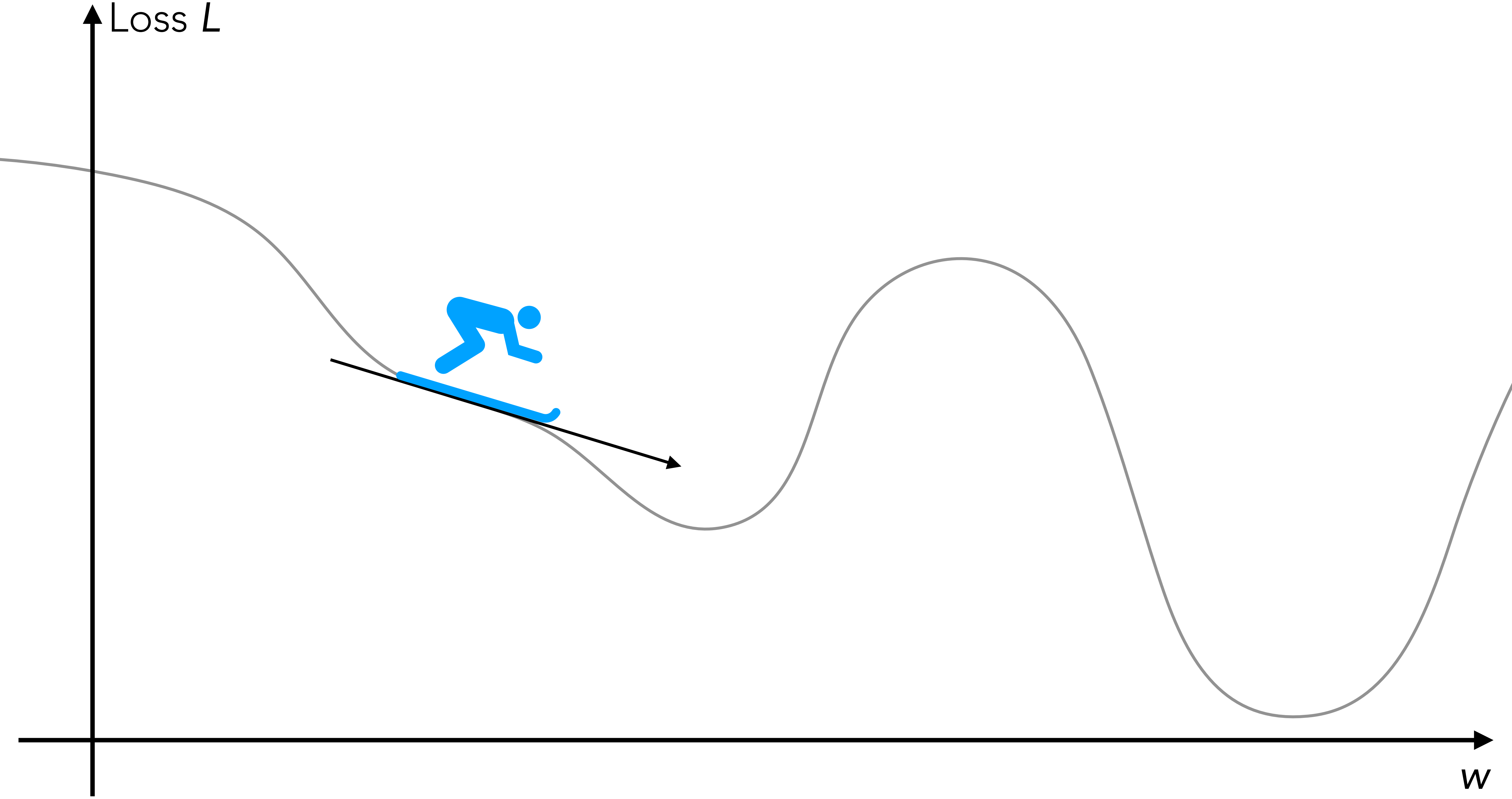
Learning a Deep Net

Suppose the neural network has a single real number parameter w



Learning a Deep Net

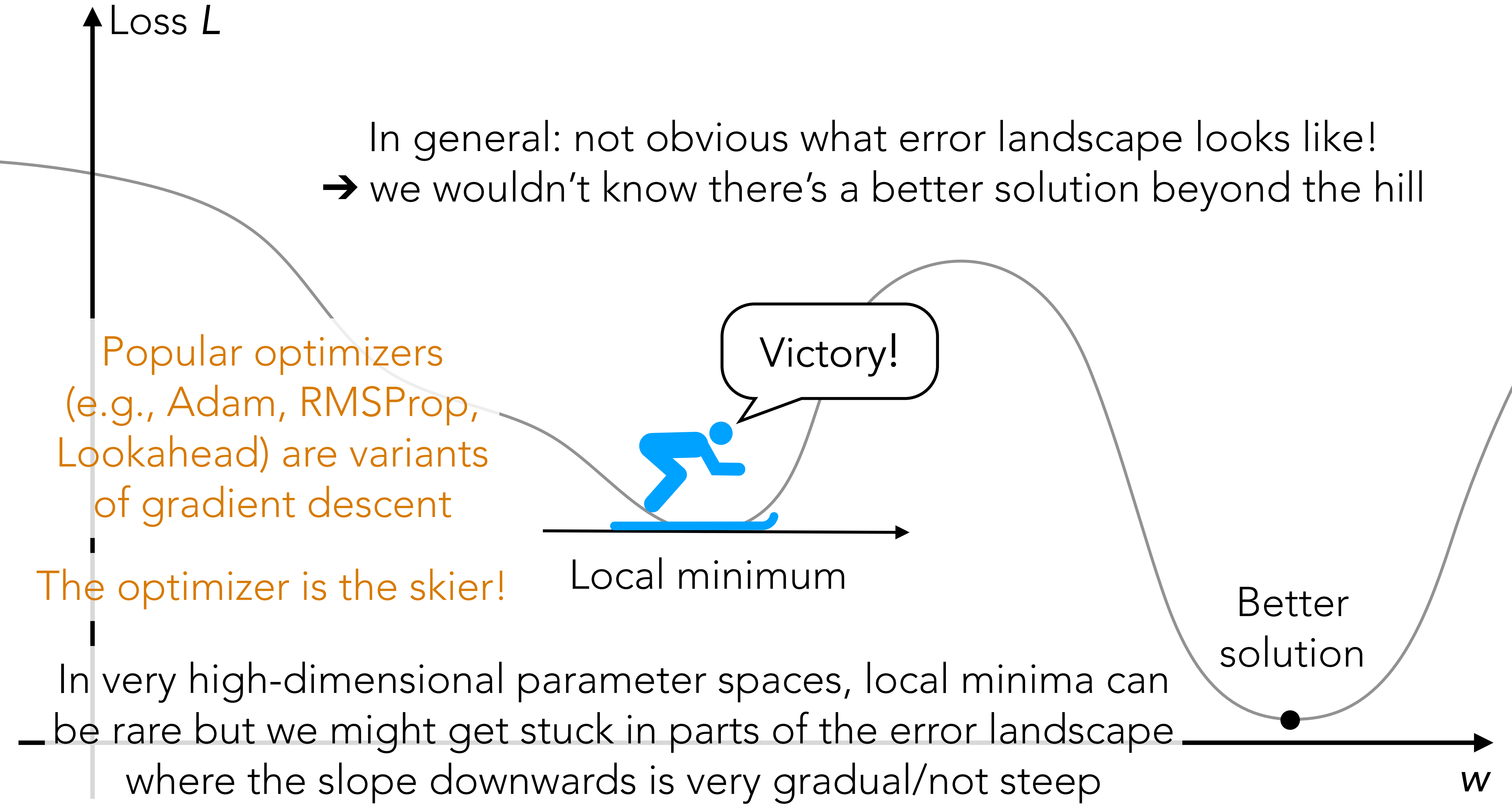
Suppose the neural network has a single real number parameter *w*



Learning a Deep Net

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

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

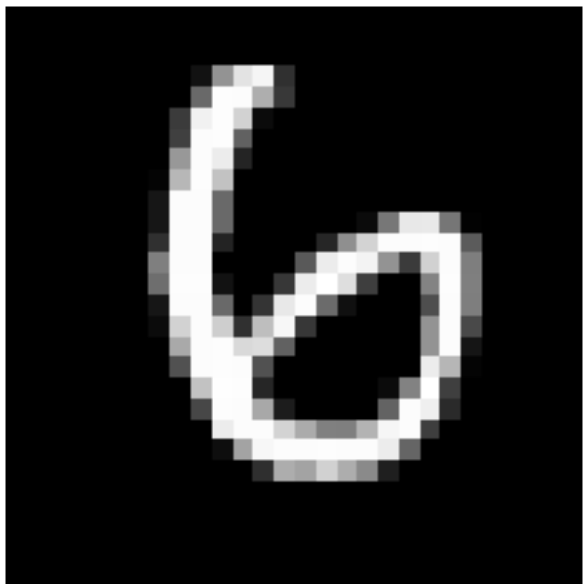
The optimizer is the skier!

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

Training label: 6

y_i



28x28 image

x_i

$f_1(x_i)$ $\quad$ $f_2(f_1(x_i))$

Loss

L

error

$L(f_2(f_1(x_i)), y_i)$

Overall loss:

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

A neural net does function composition!

f_1 $\quad$ f_2

All parameters: θ

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

Automatic differentiation is crucial in learning deep nets!

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

Gradient Descent

Training example 1	Training example 2	Training example 3	Training example 4	Training example 5	…	Training example *n*
loss 1	loss 2	loss 3	loss 4	loss 5	…	loss *n*

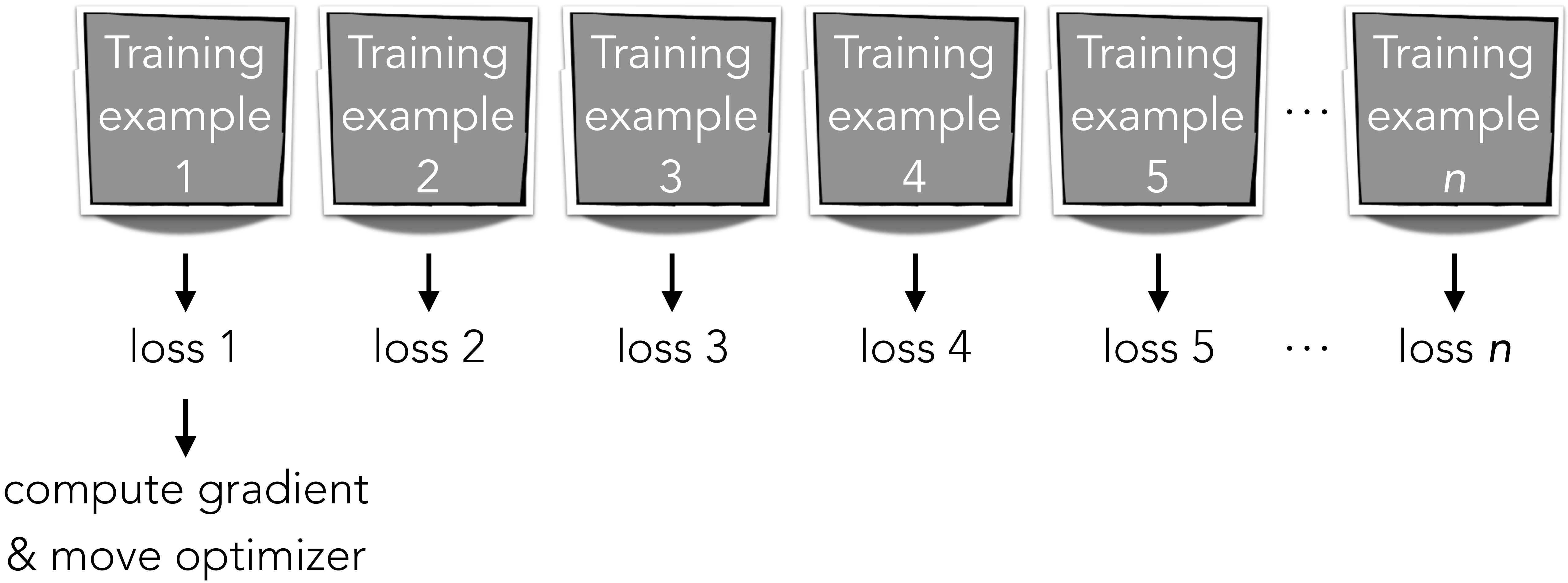
average loss

compute gradient
& move optimizer

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

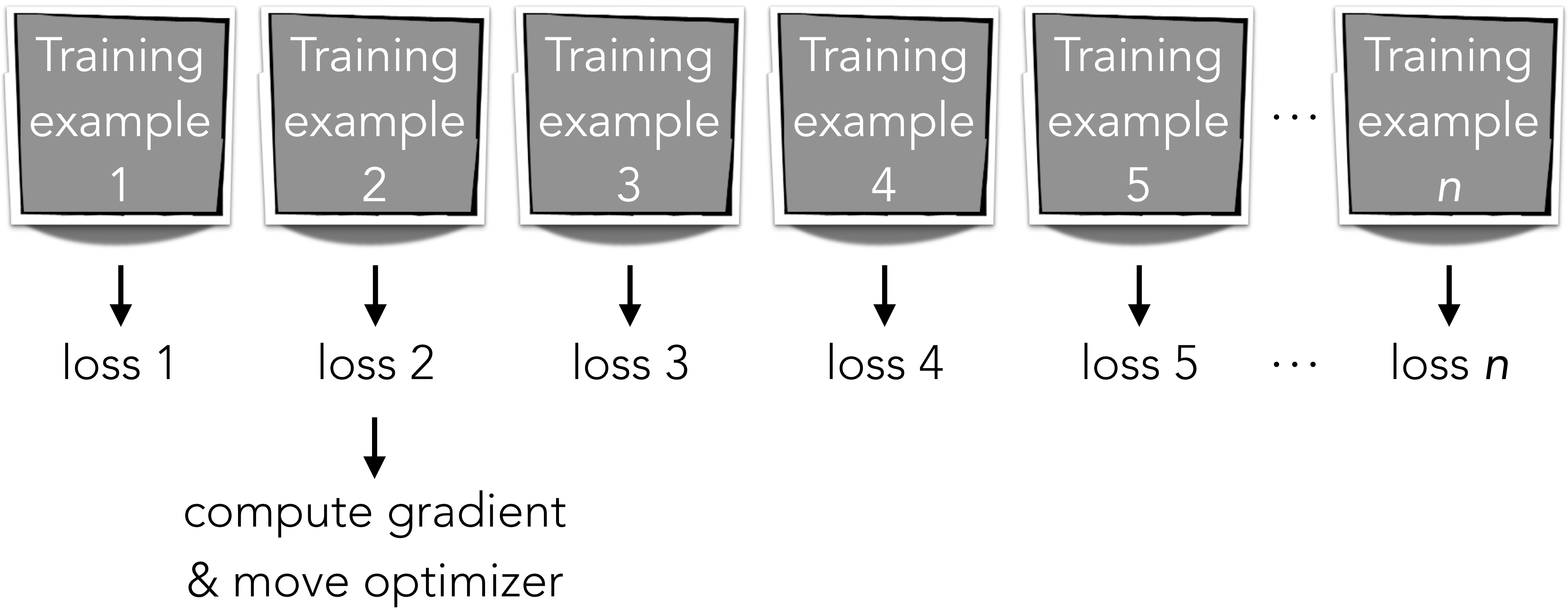
Computing gradients using all the training data seems really expensive!

Stochastic Gradient Descent (SGD)



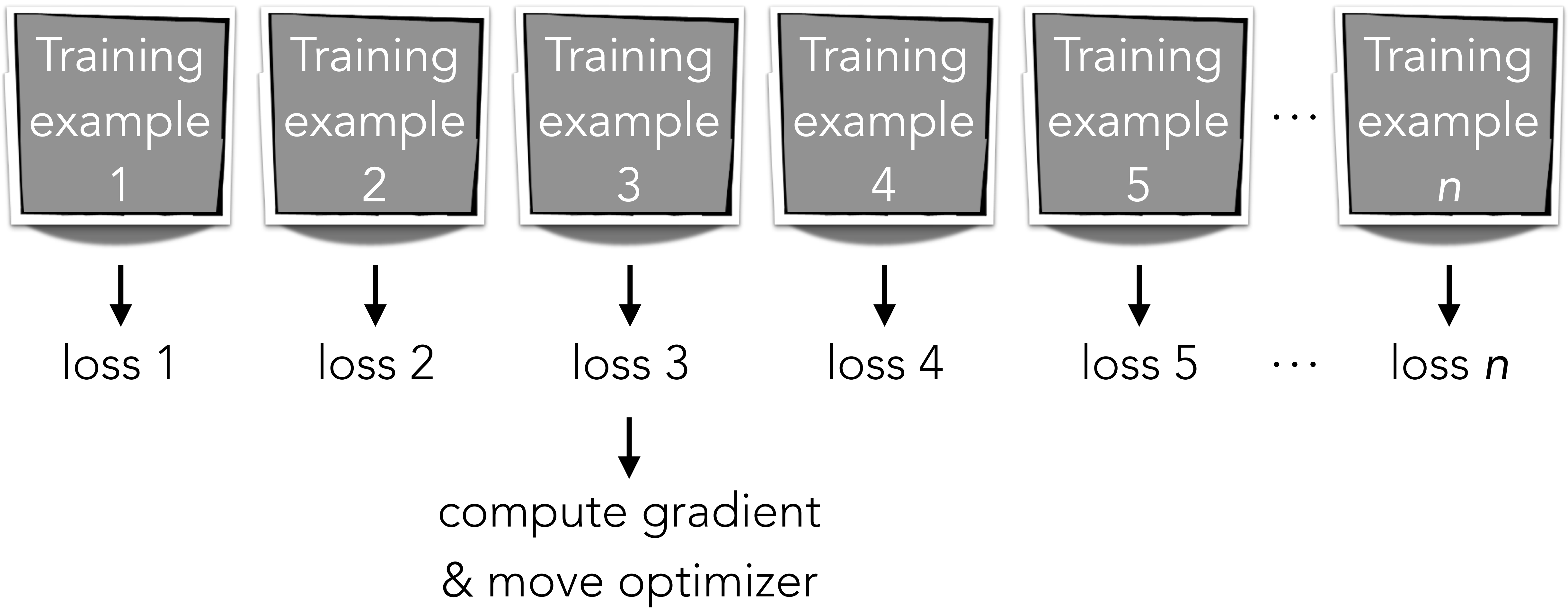
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the “full” gradient)

Stochastic Gradient Descent (SGD)



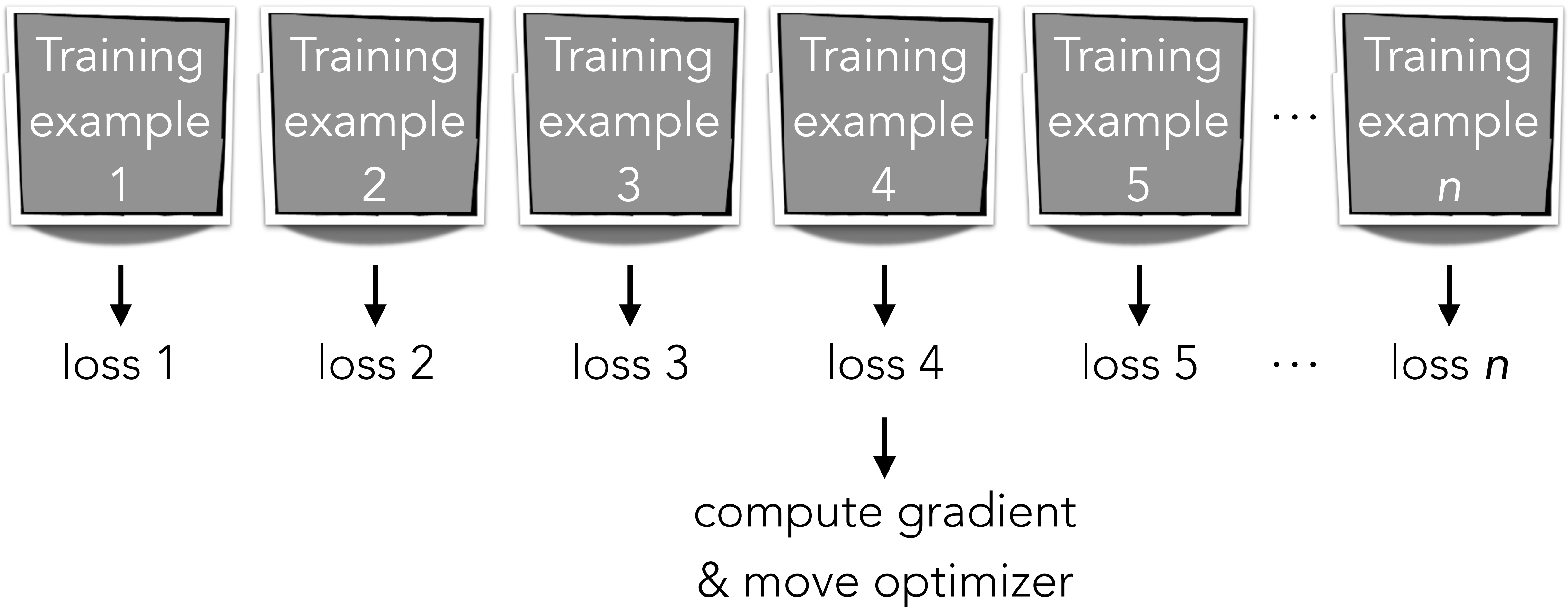
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



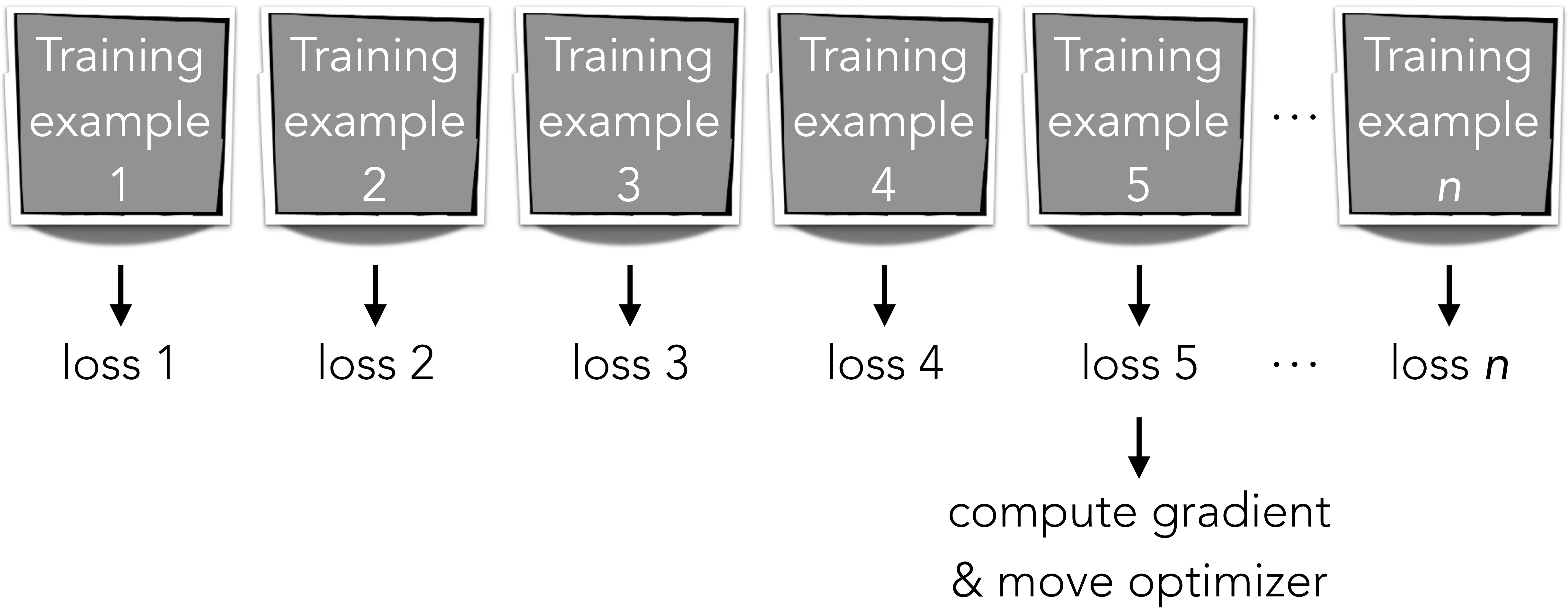
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



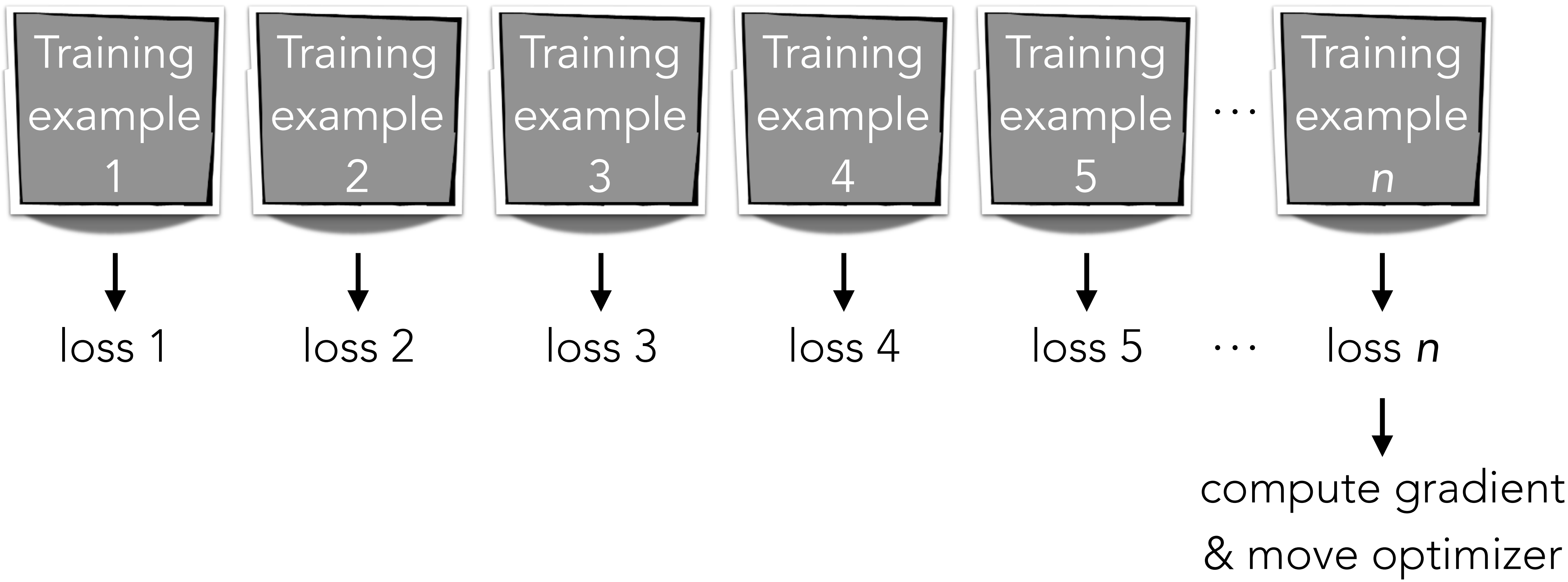
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

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)

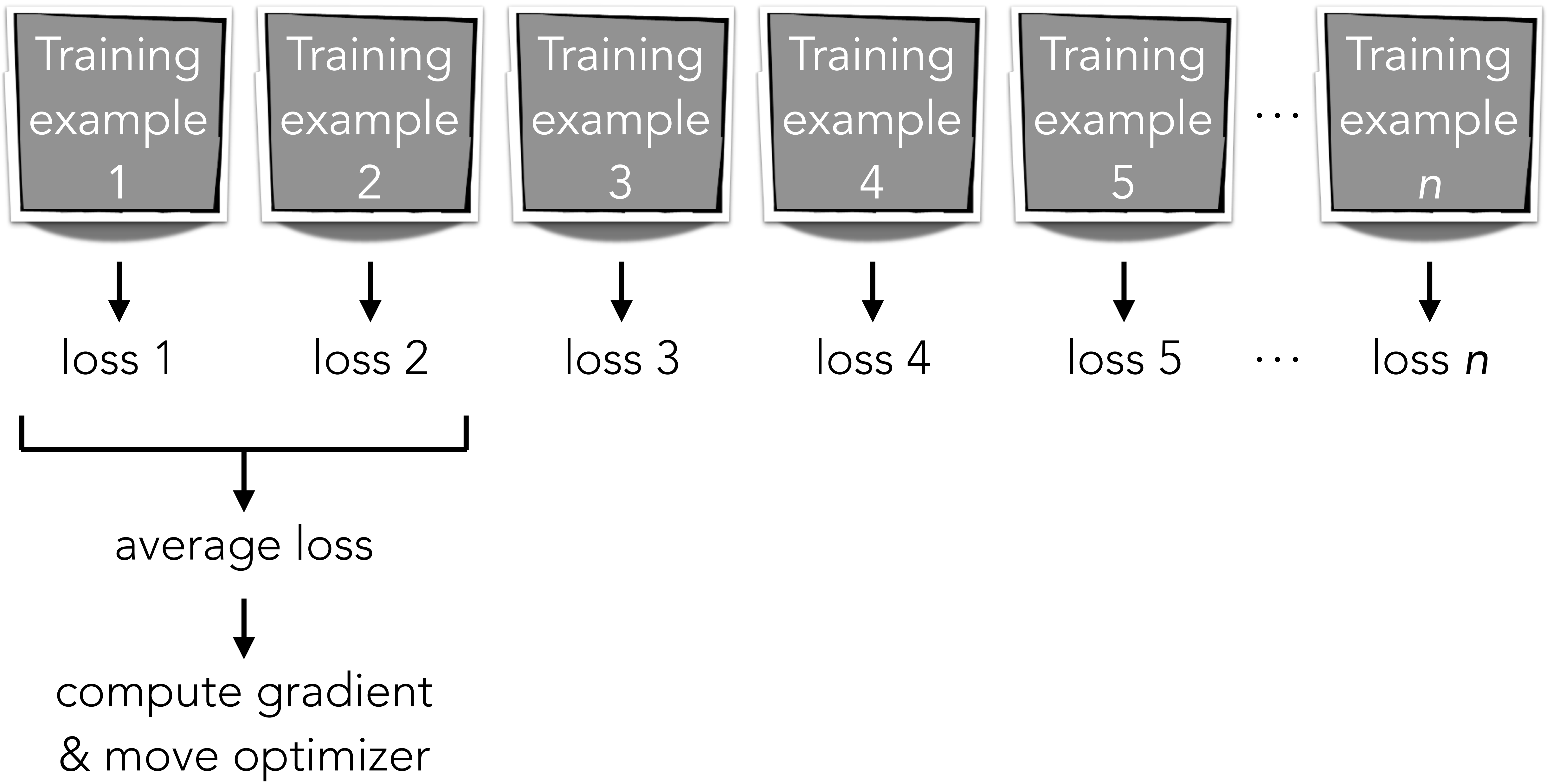
Stochastic Gradient Descent (SGD)

Training example 1	Training example 2	Training example 3	Training example 4	Training example 5	…	Training example *n*
↓	↓	↓	↓	↓		↓
loss 1	loss 2	loss 3	loss 4	loss 5	…	loss *n*
↓						
compute gradient & move optimizer						

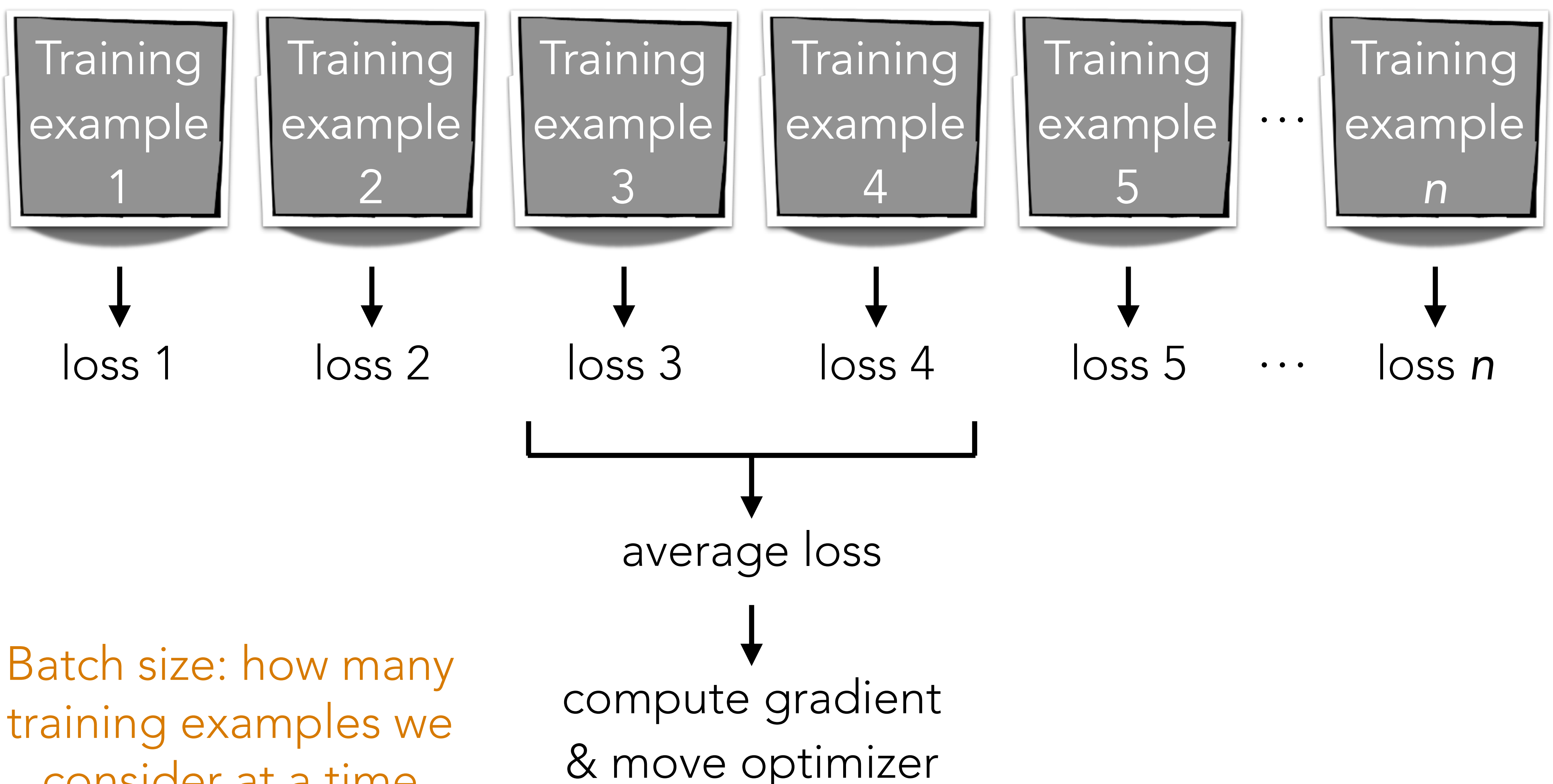
An epoch refers to 1 full pass through all the training data

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



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?

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`

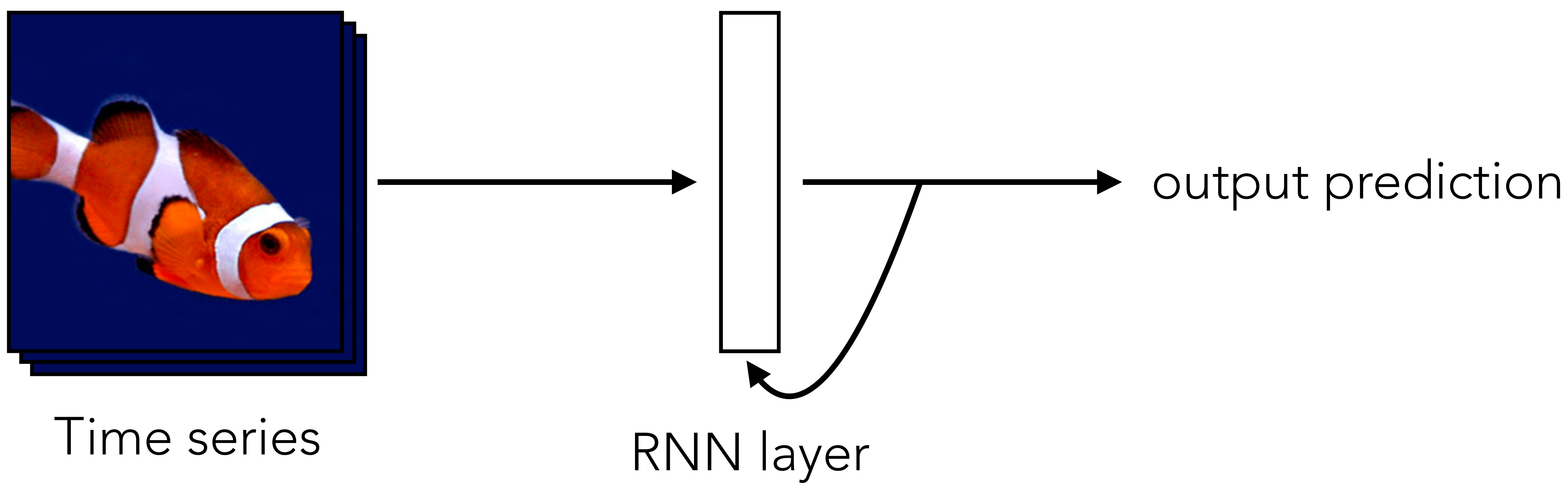
A special kind of RNN: an “LSTM”

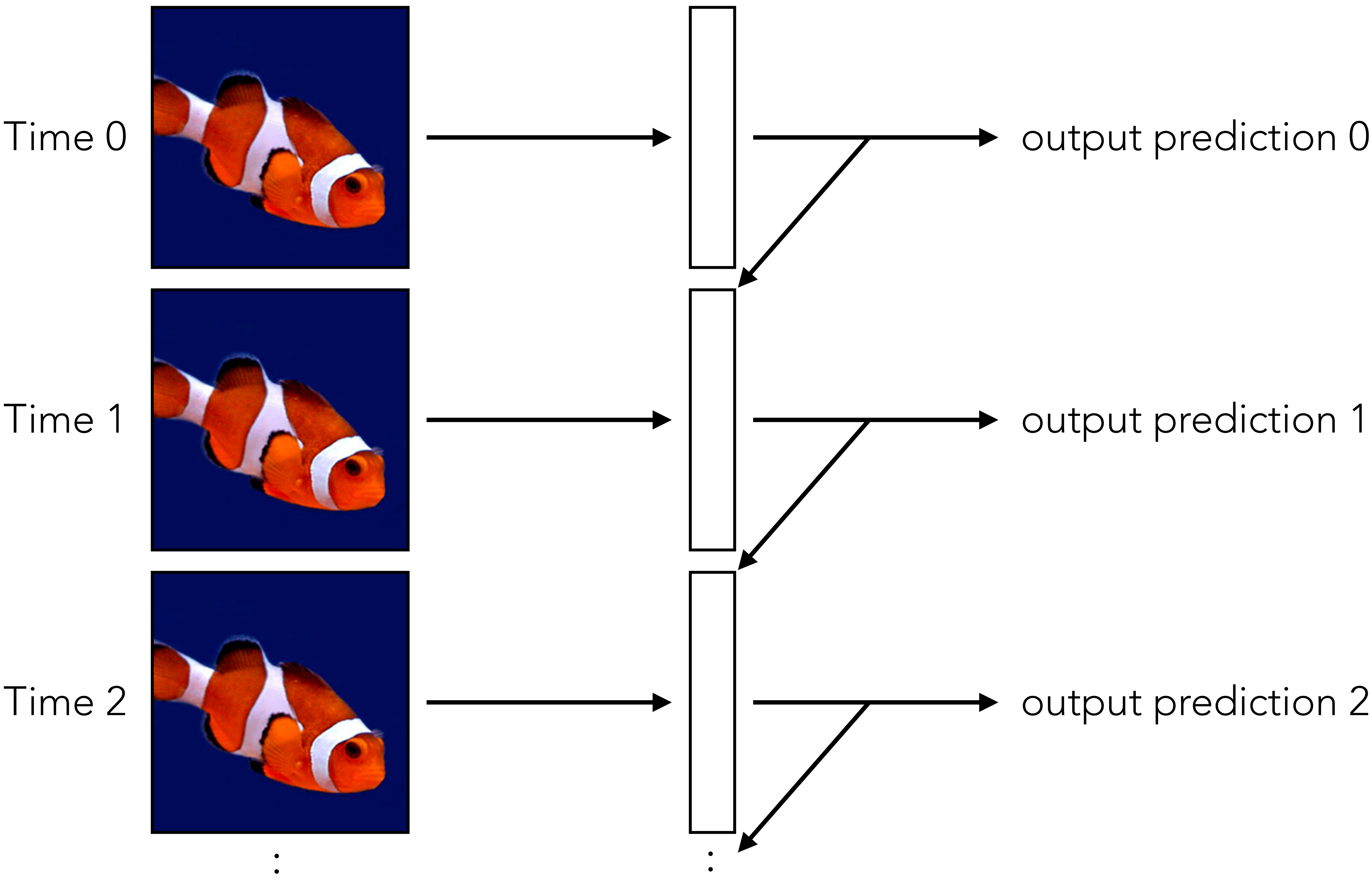
(Flashback) Vanilla ReLU RNN

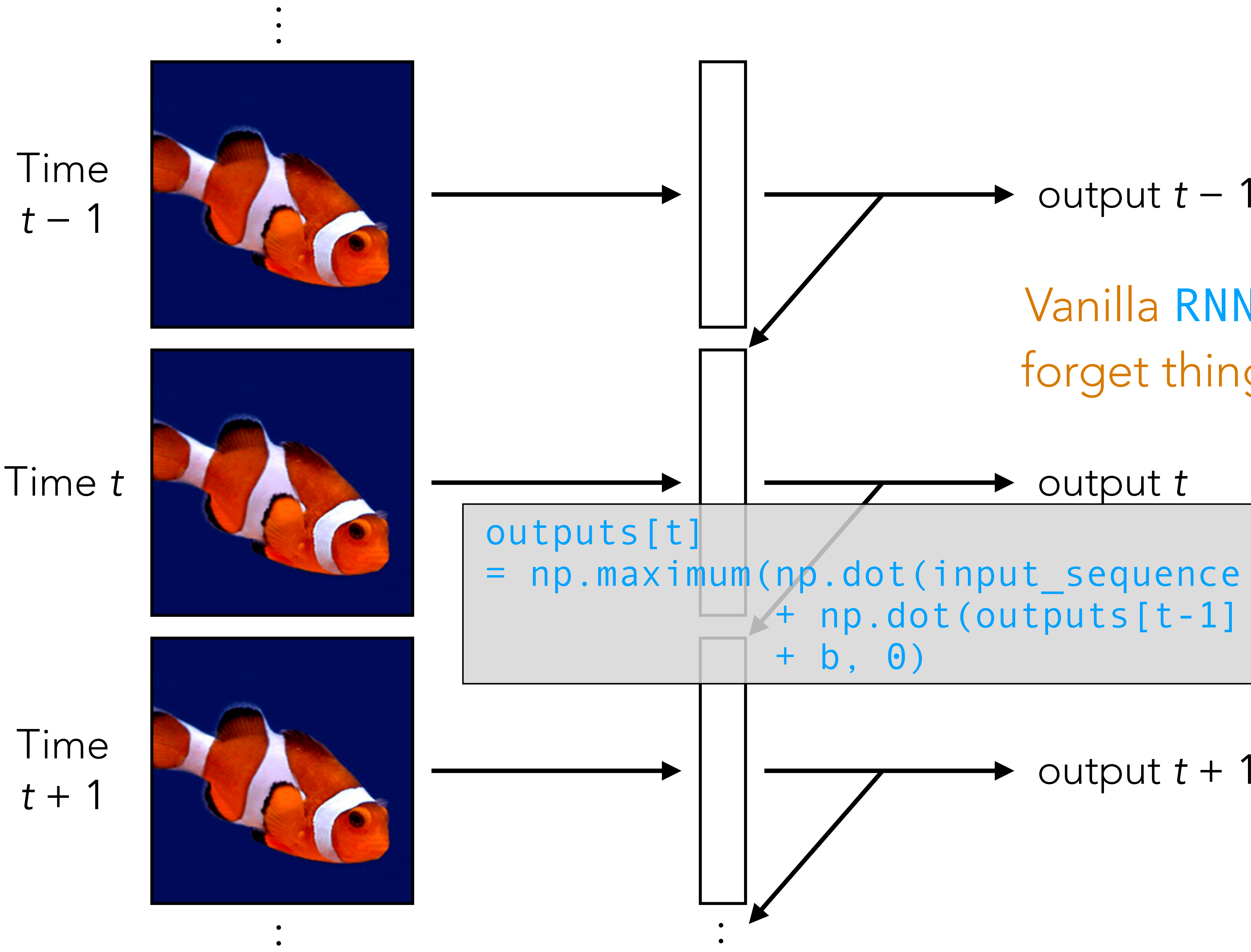
```
current_state = np.zeros(num_nodes)
outputs = []
for input in input_sequence:
  linear = np.dot(input, W.T) + b    \
         + np.dot(current_state, U.T)
  output = np.maximum(0, linear) # ReLU
  outputs.append(output)
  current_state = output
```

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

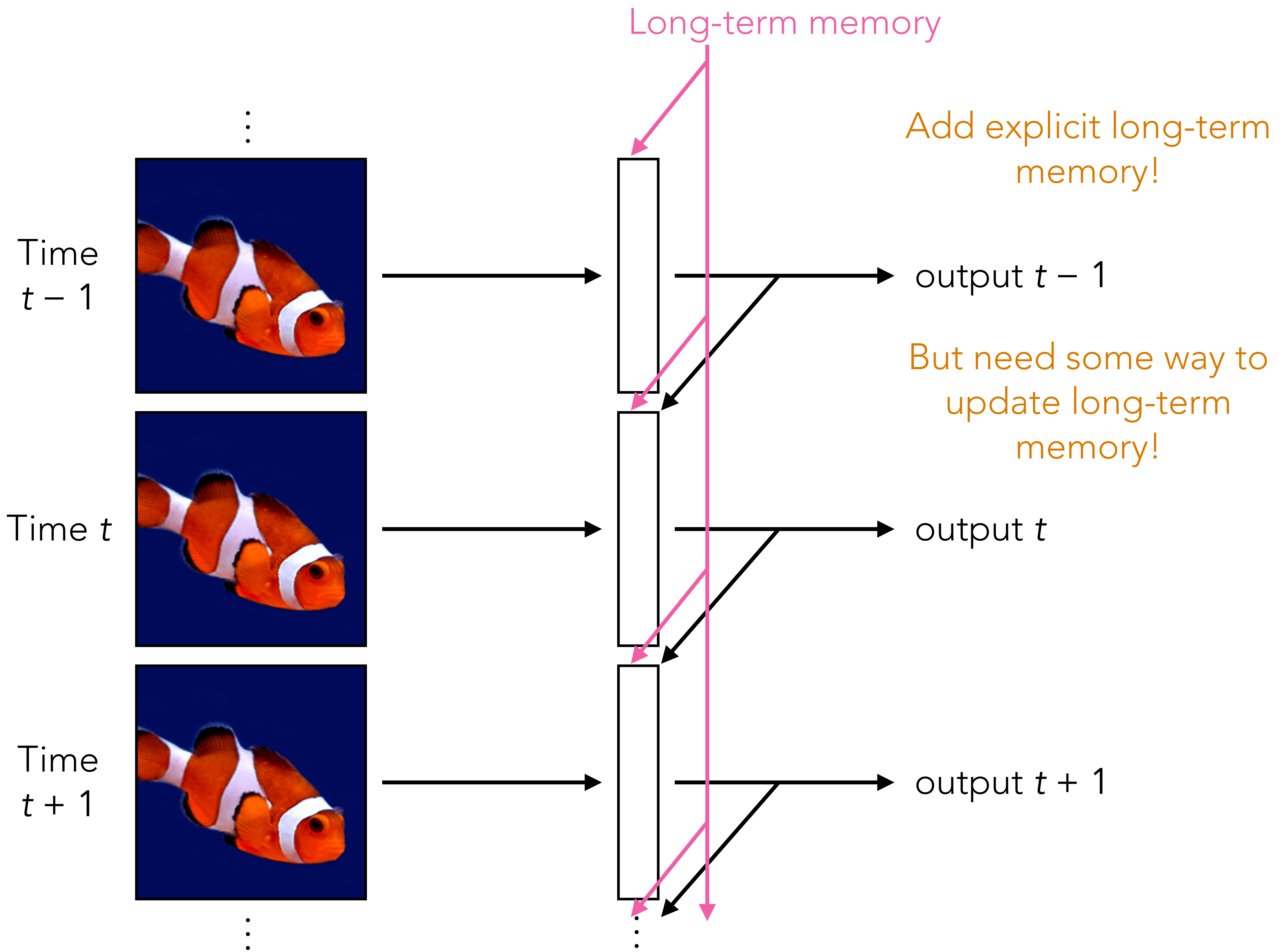

Time series
RNN layer
output prediction

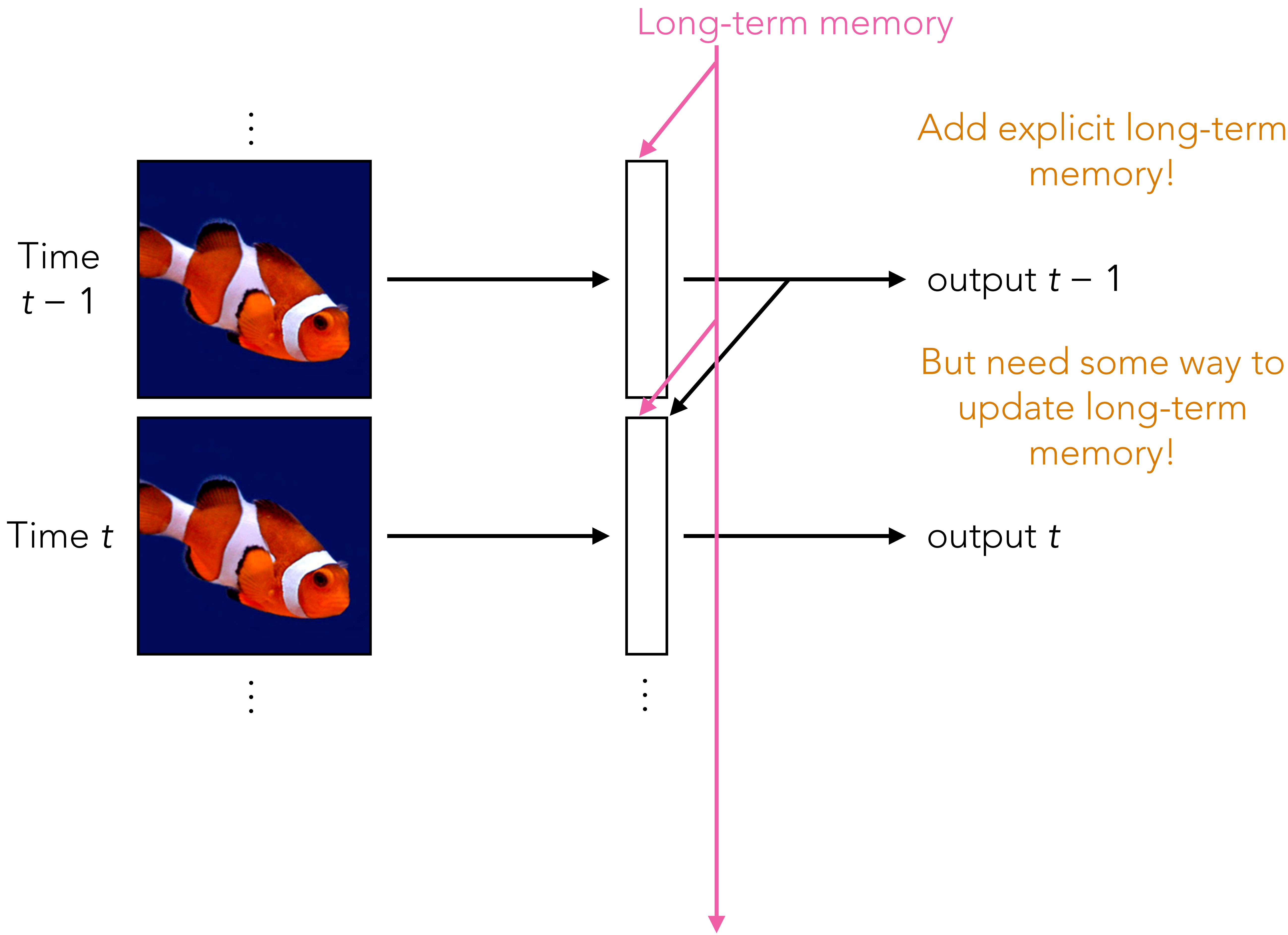

Time 0
output prediction 0
Time 1
output prediction 1
Time 2
output prediction 2

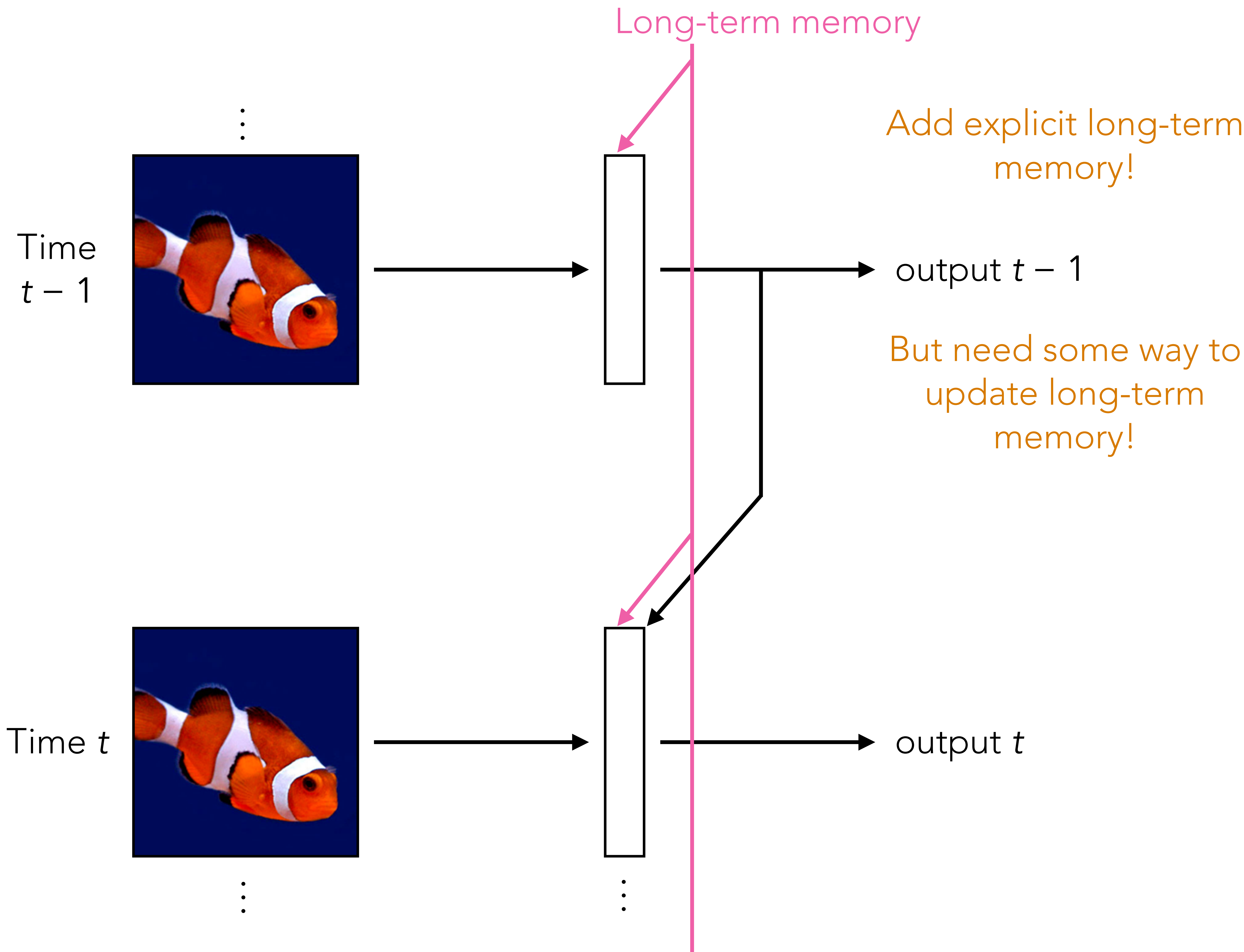


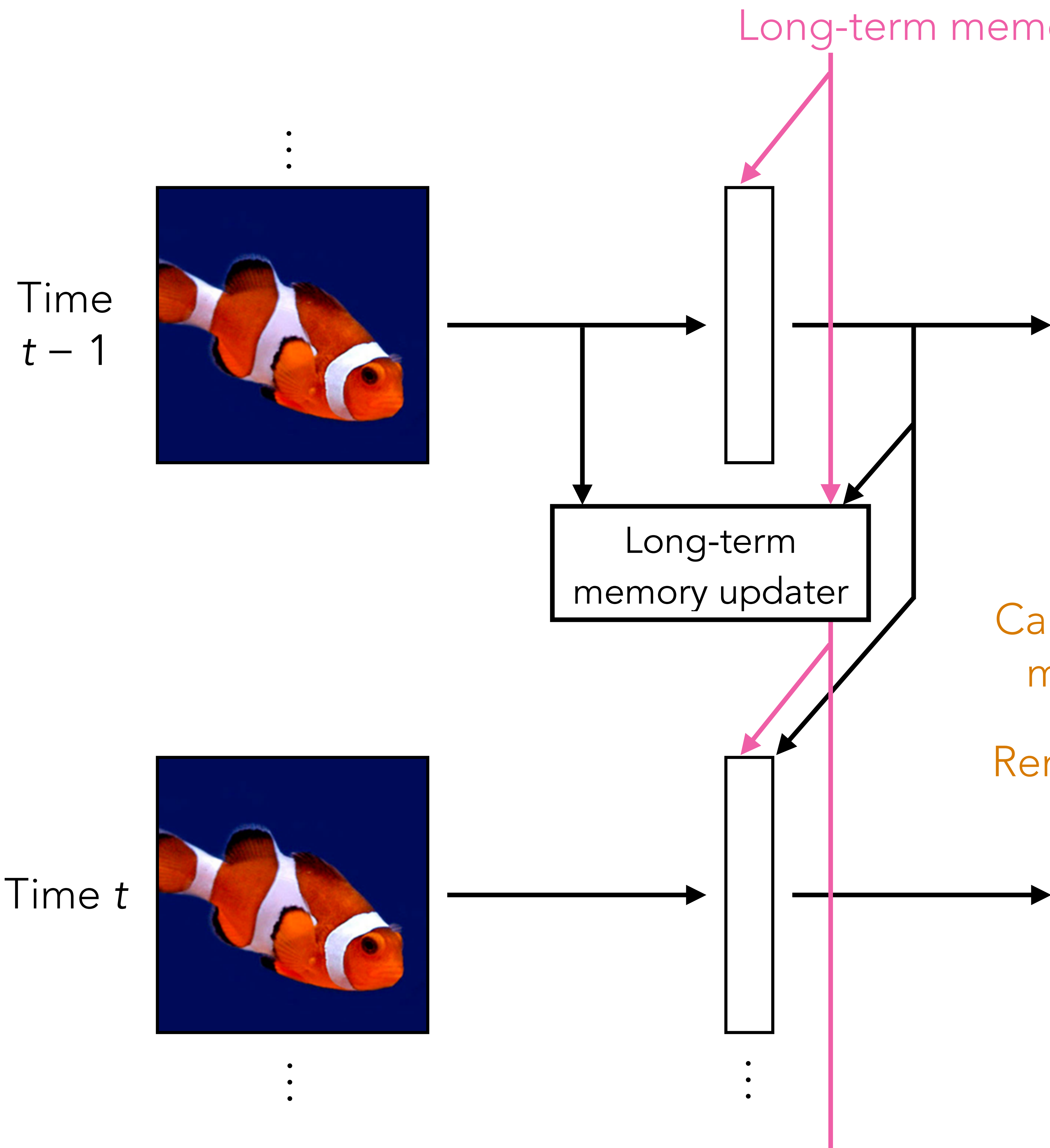
Vanilla RNN tends to forget things quickly

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


Long-term memory
Add explicit long-term memory!
But need some way to update 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


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

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 all have the same length (same number of time steps) and 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
- Note: while it is possible to have a CNN take in inputs that vary in size, we did not cover this in lecture

Full Transformer

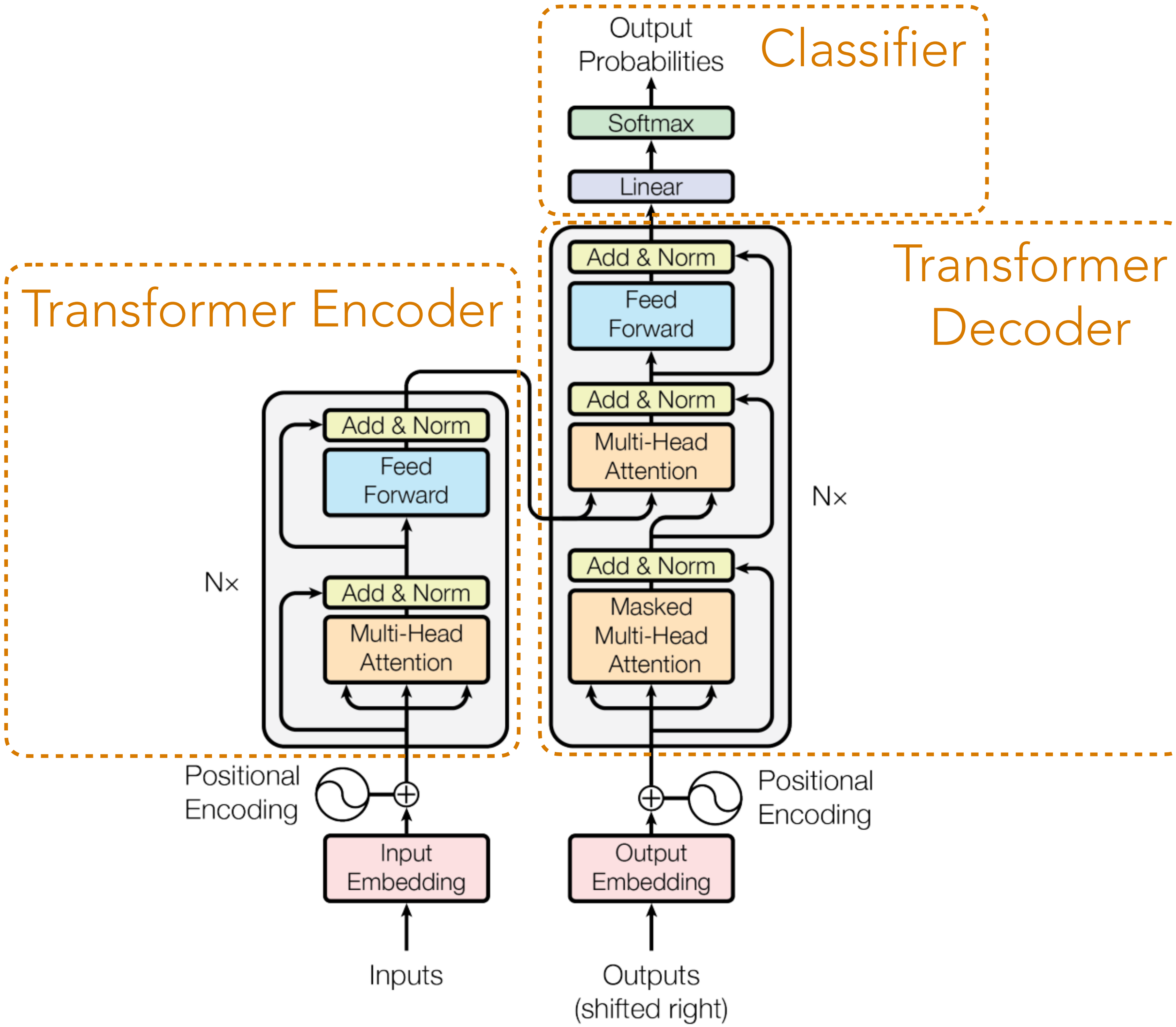


Figure 1: The Transformer - model architecture.

Vaswani et al. “Attention is All You Need”. NeurIPS 2017.

Decoder-Only Transformer

The feed forward network used is just an MLP

"Norm" refers to LayerNorm

"Masked" just is a reference to the causal dependency enforced (current time step's output cannot depend on future time step's inputs)

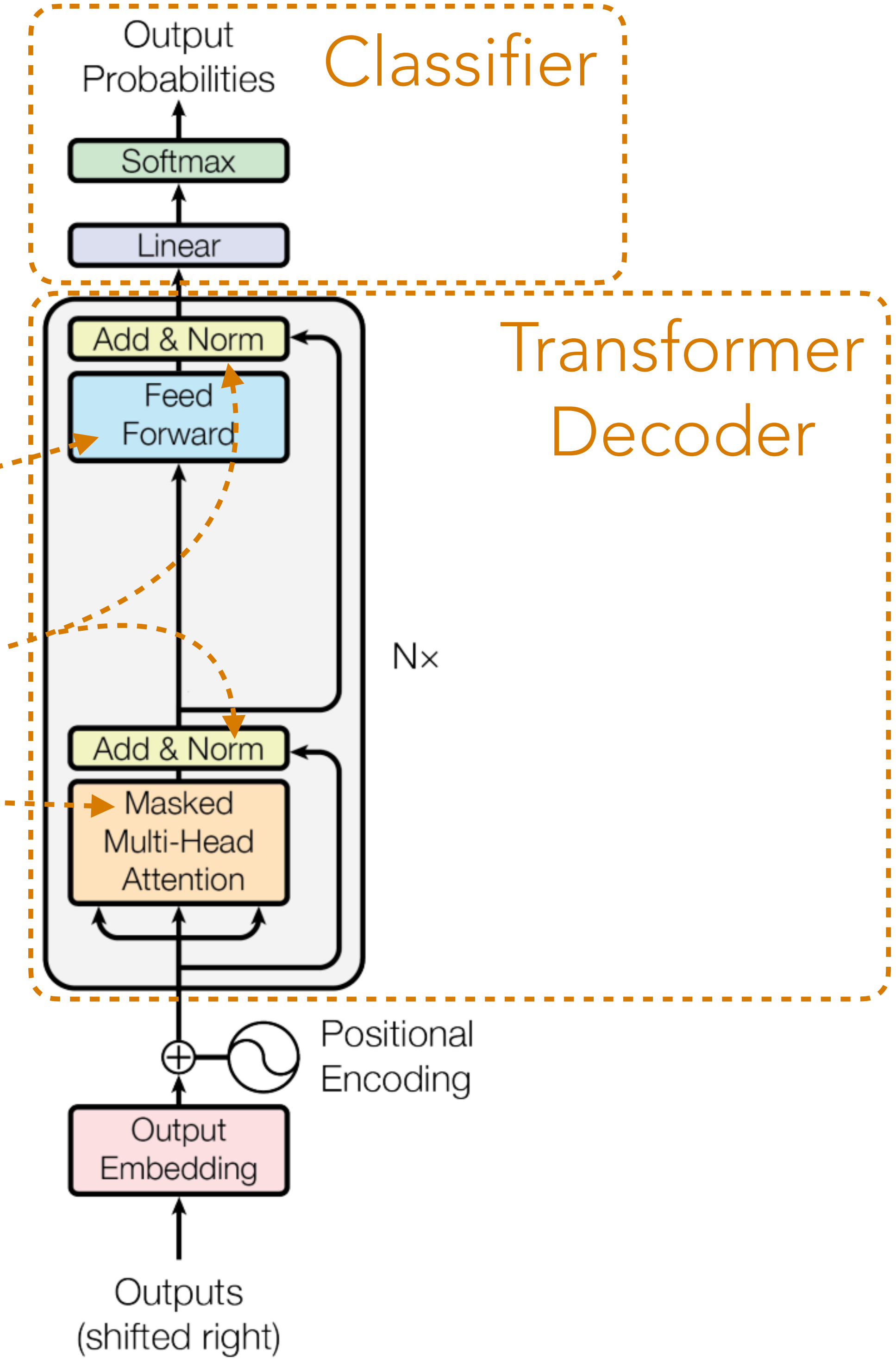


Figure 1: The Transformer - model architecture.

Vaswani et al. "Attention is All You Need". NeurIPS 2017.

Decoder-Only Transformer

The feed forward network used is just an MLP

"Norm" refers to LayerNorm

"Masked" just is a reference to the causal dependency enforced (current time step's output cannot depend on future time step's inputs)

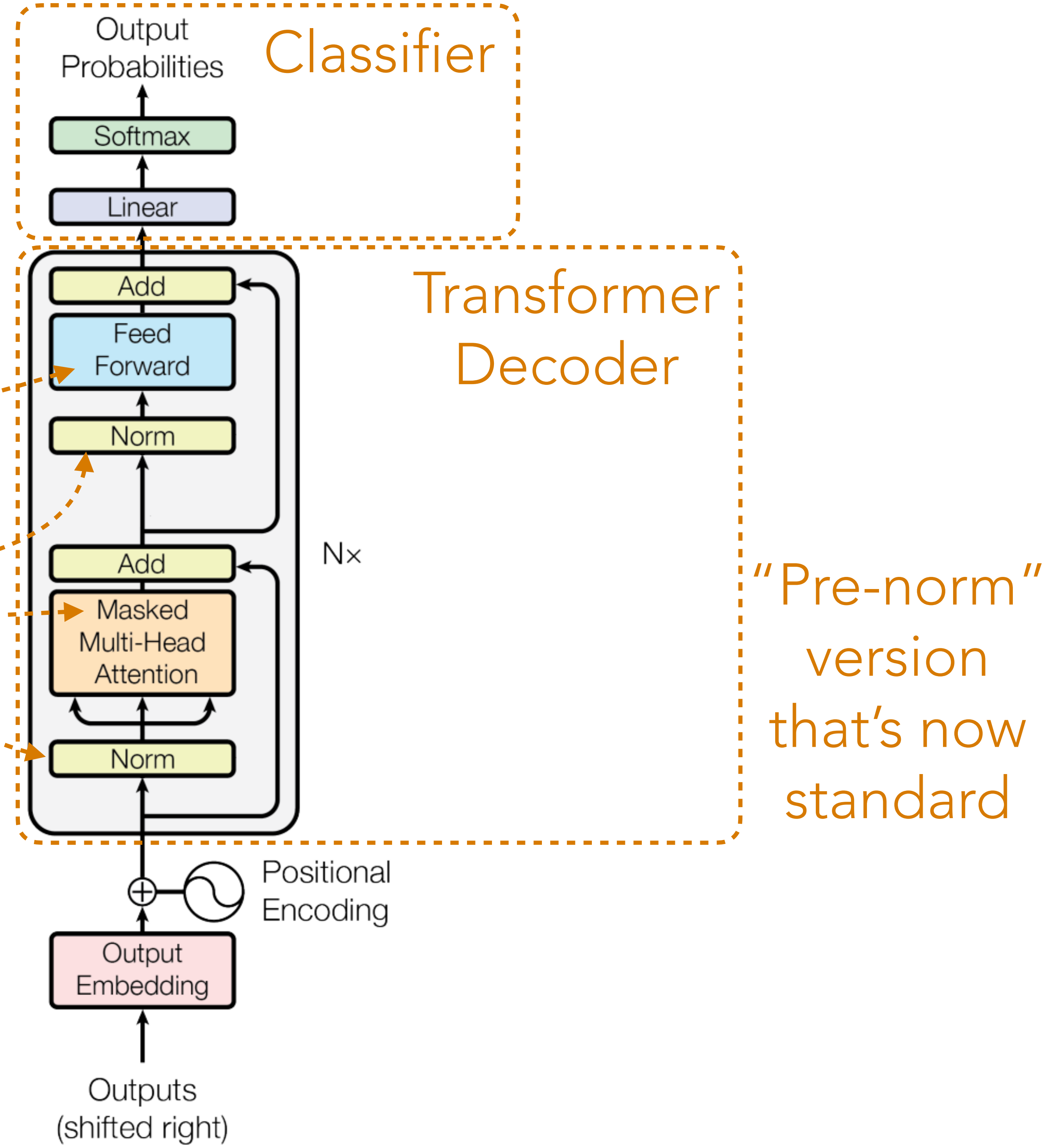


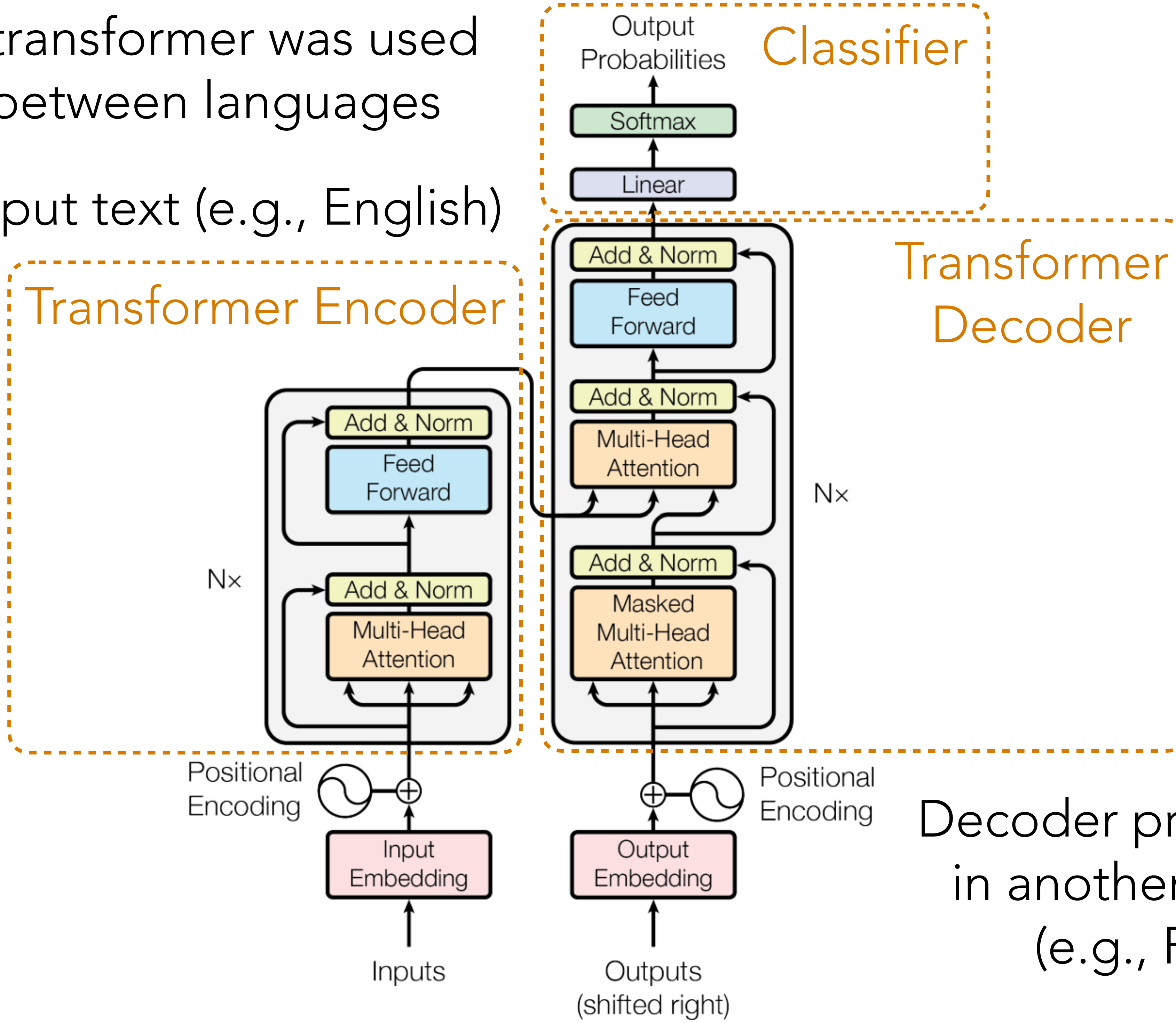
Figure 1: The Transformer - model architecture.

Vaswani et al. "Attention is All You Need". NeurIPS 2017.

Full Transformer

The original full transformer was used for translating between languages

Encoder sees input text (e.g., English)



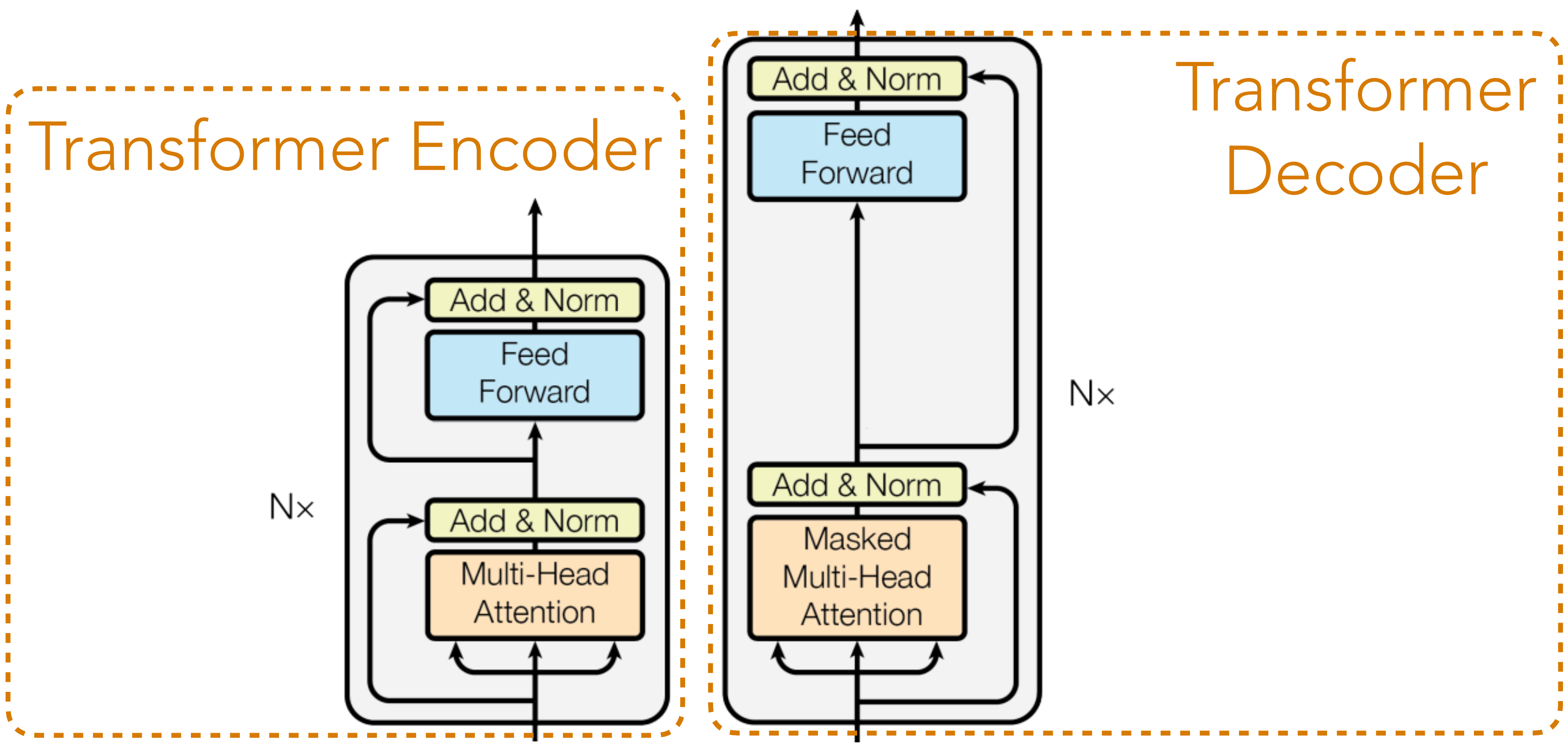
Decoder produces text in another language (e.g., French)

Figure 1: The Transformer - model architecture.

Vaswani et al. "Attention is All You Need". NeurIPS 2017.

Transformer Encoder vs Transformer Decoder

In PyTorch, `TransformerEncoder` allows the user to specify a causal mask, which would turn it into a transformer decoder



The only difference is the causal masking

Meanwhile, if you use PyTorch's `TransformerDecoder`, it expects that you provide it information from the encoder…which we wouldn't have if we're using a decoder-only transformer so that's why the lecture code demo just uses the `TransformerEncoder` with a causal mask…

Questions About the Lecture Demo?

Demo