Carnegie Mellon University
Heinz College

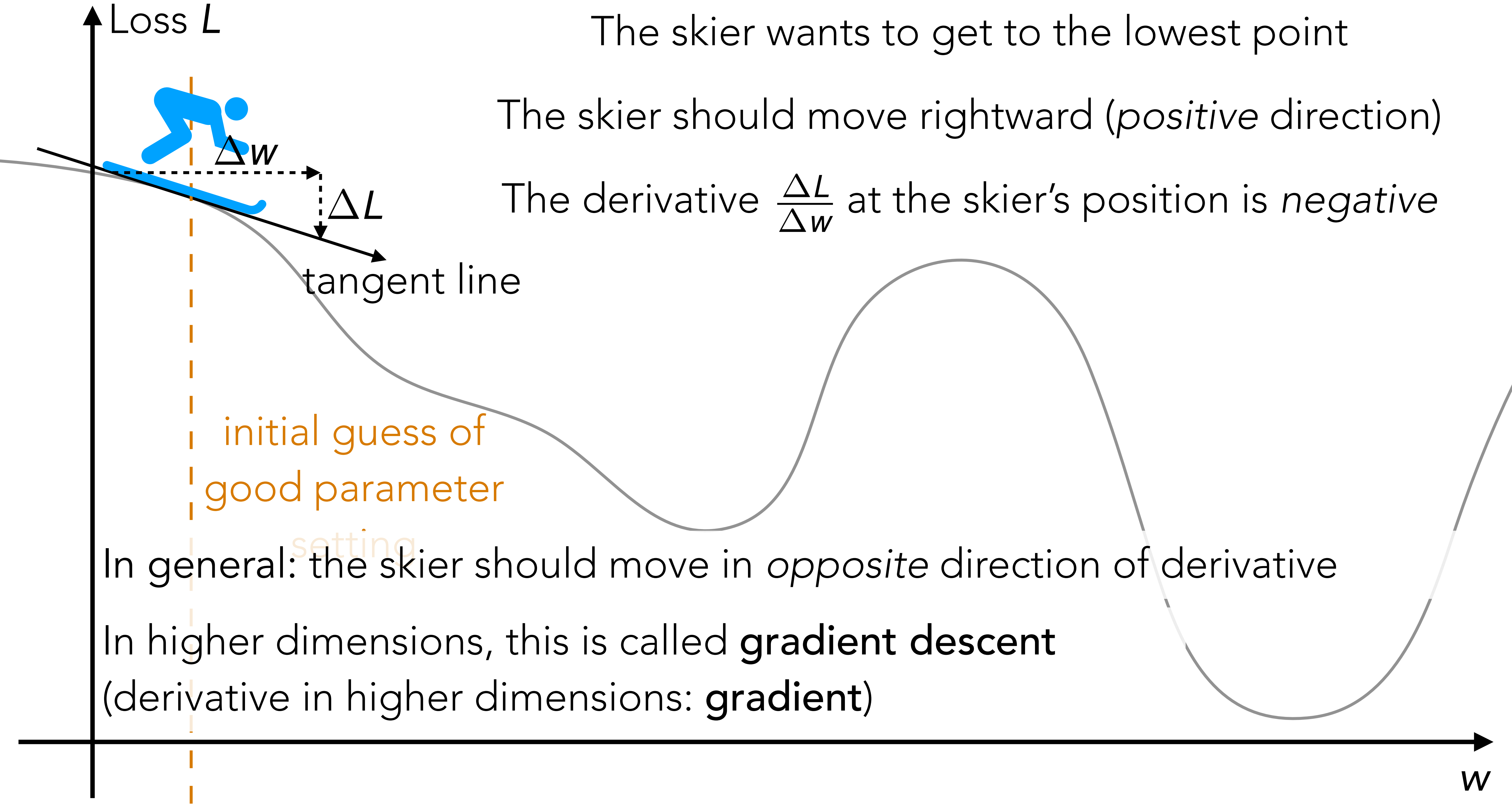
95-865 Unstructured Data Analytics

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)

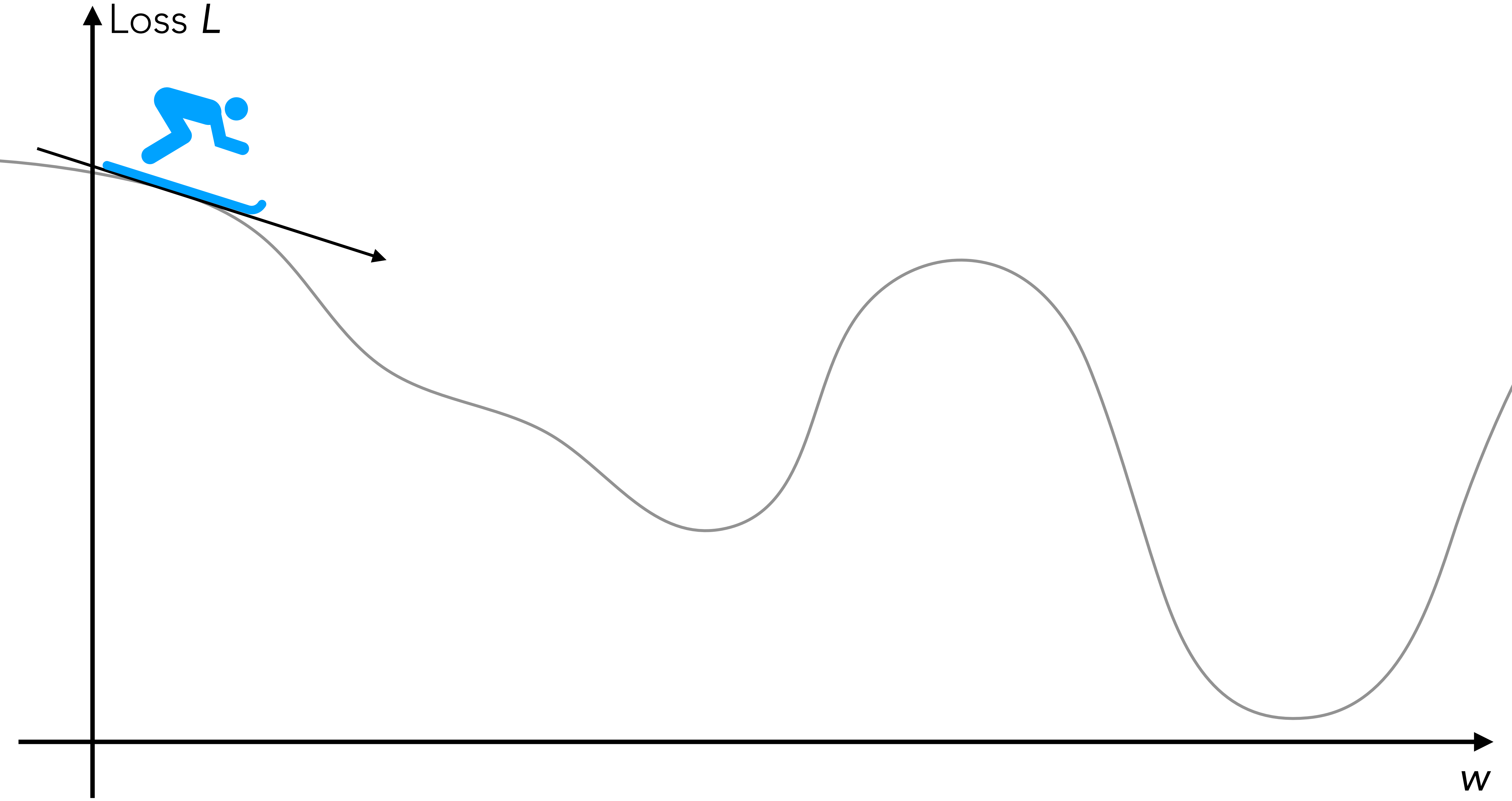
The derivative $\frac{\Delta L}{\Delta w}$ at the skier's position is *negative*

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

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

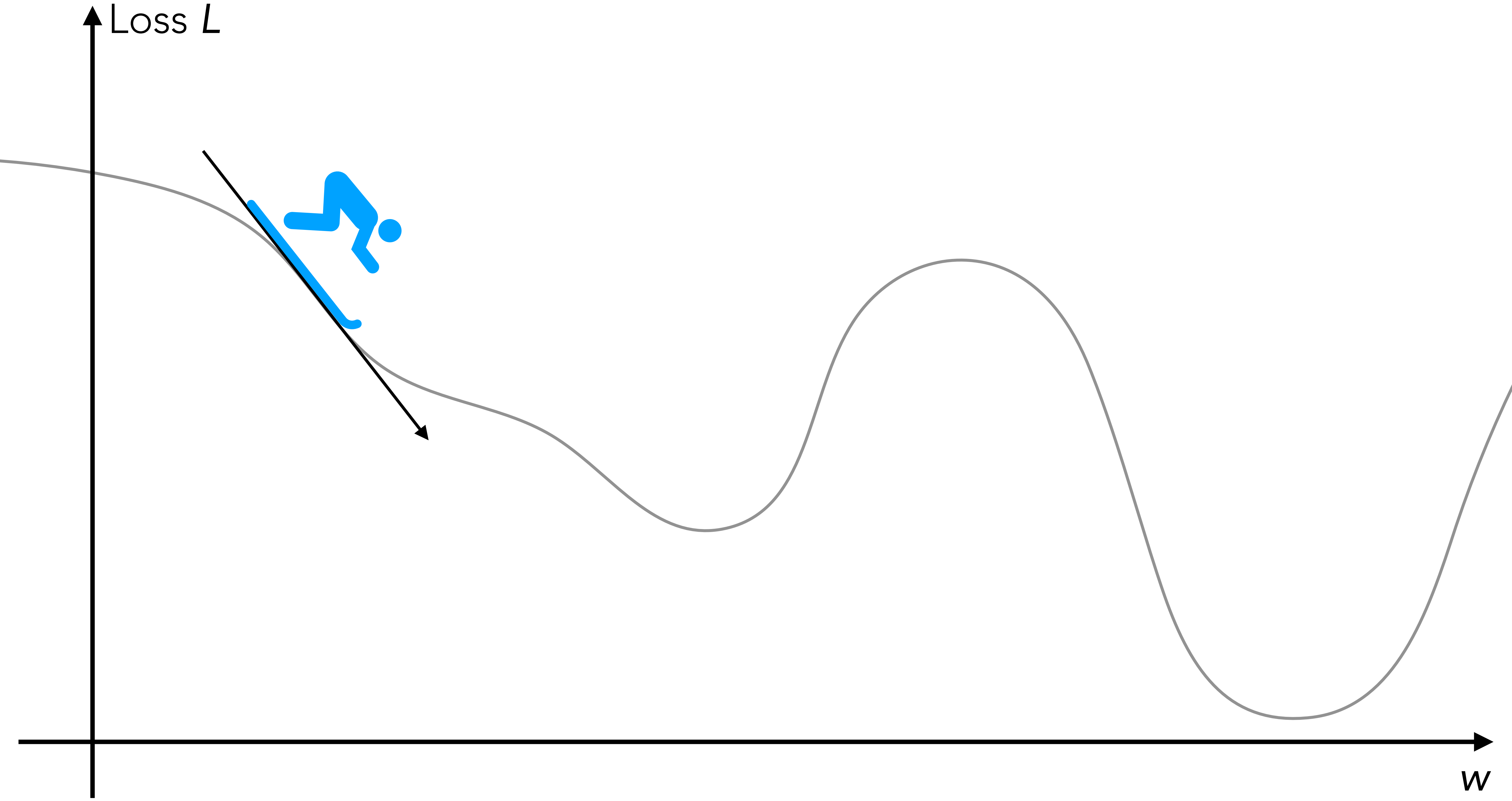
Learning a Deep Net

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



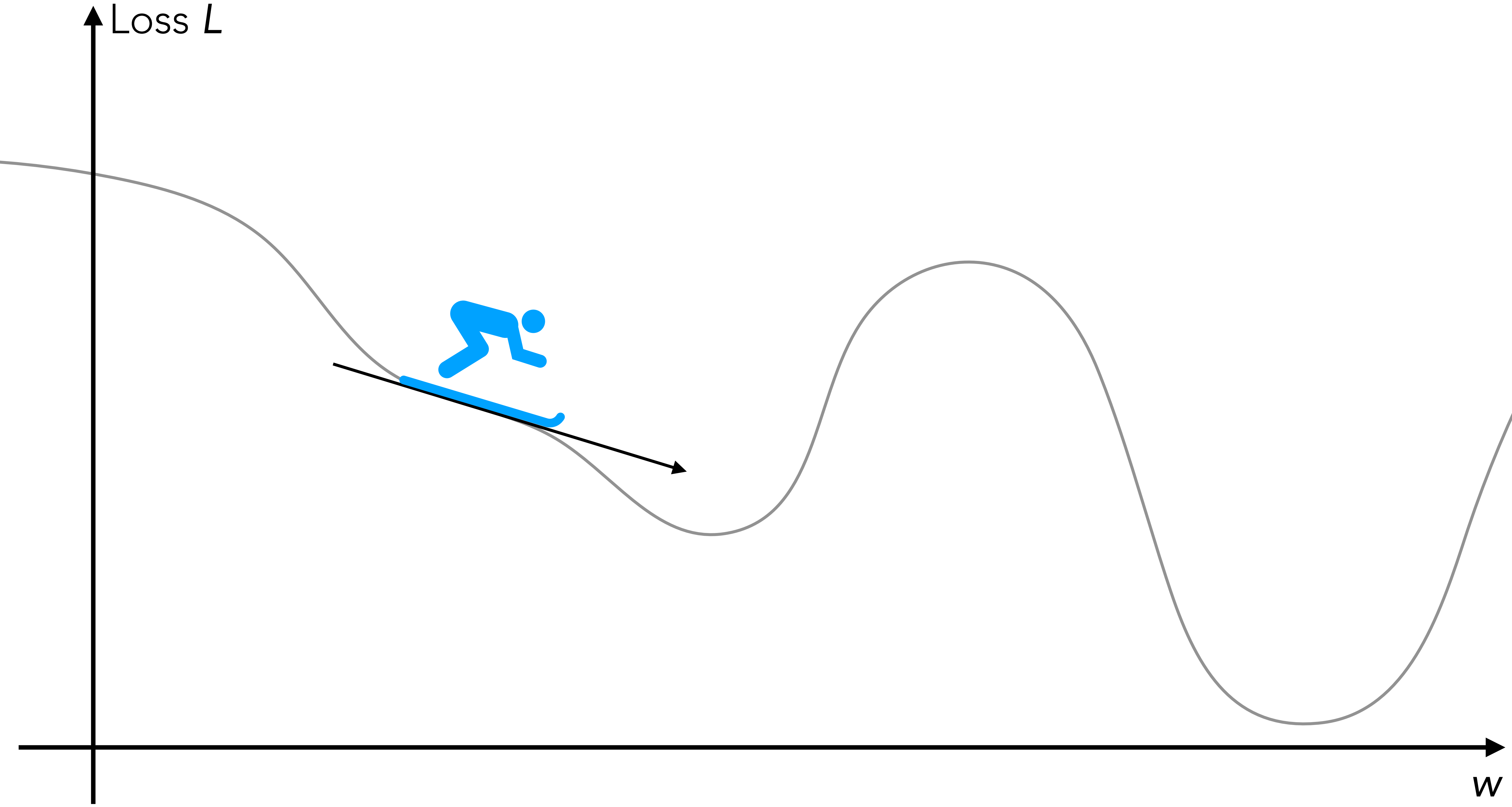
Learning a Deep Net

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



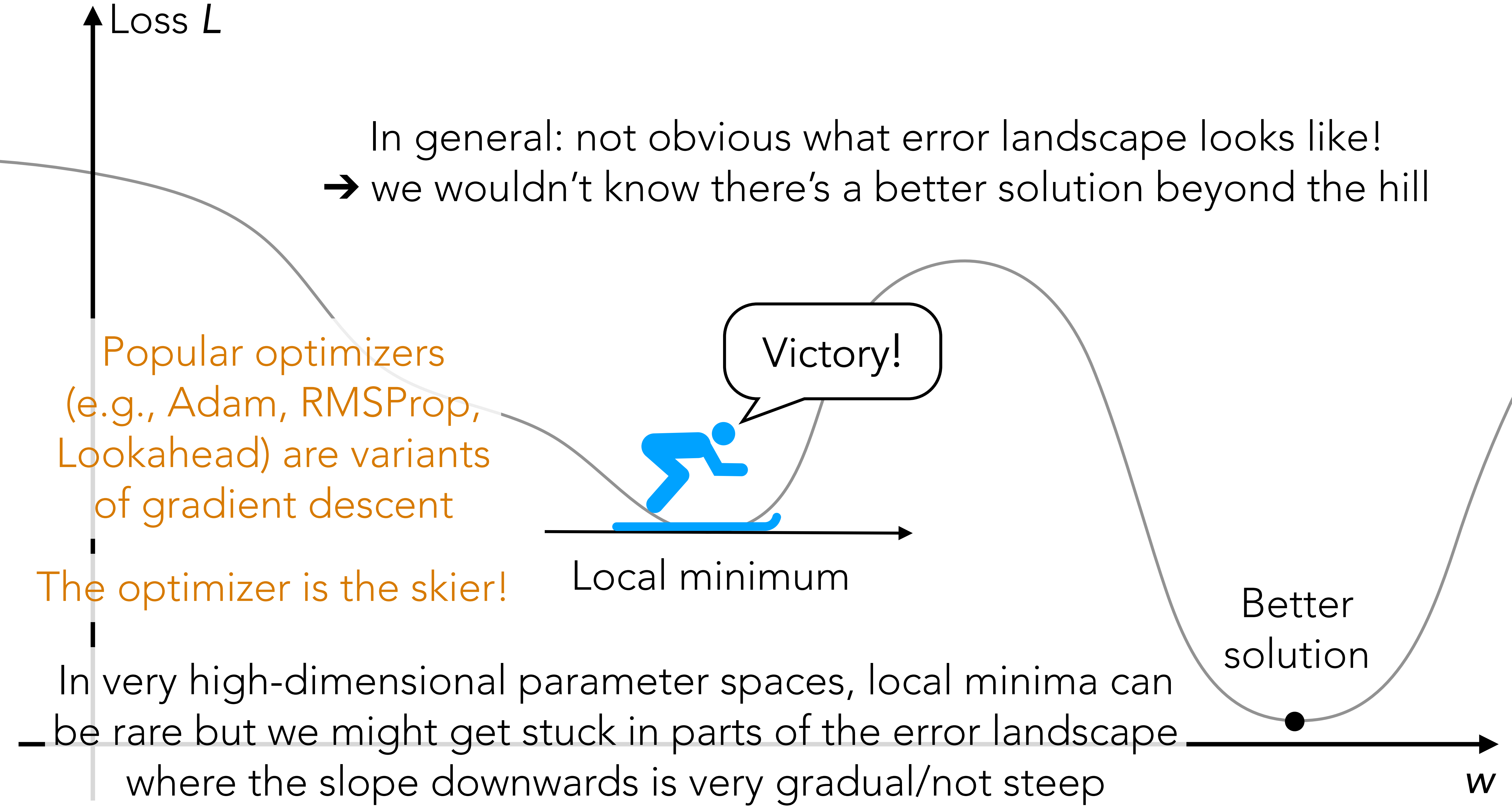
Learning a Deep Net

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*



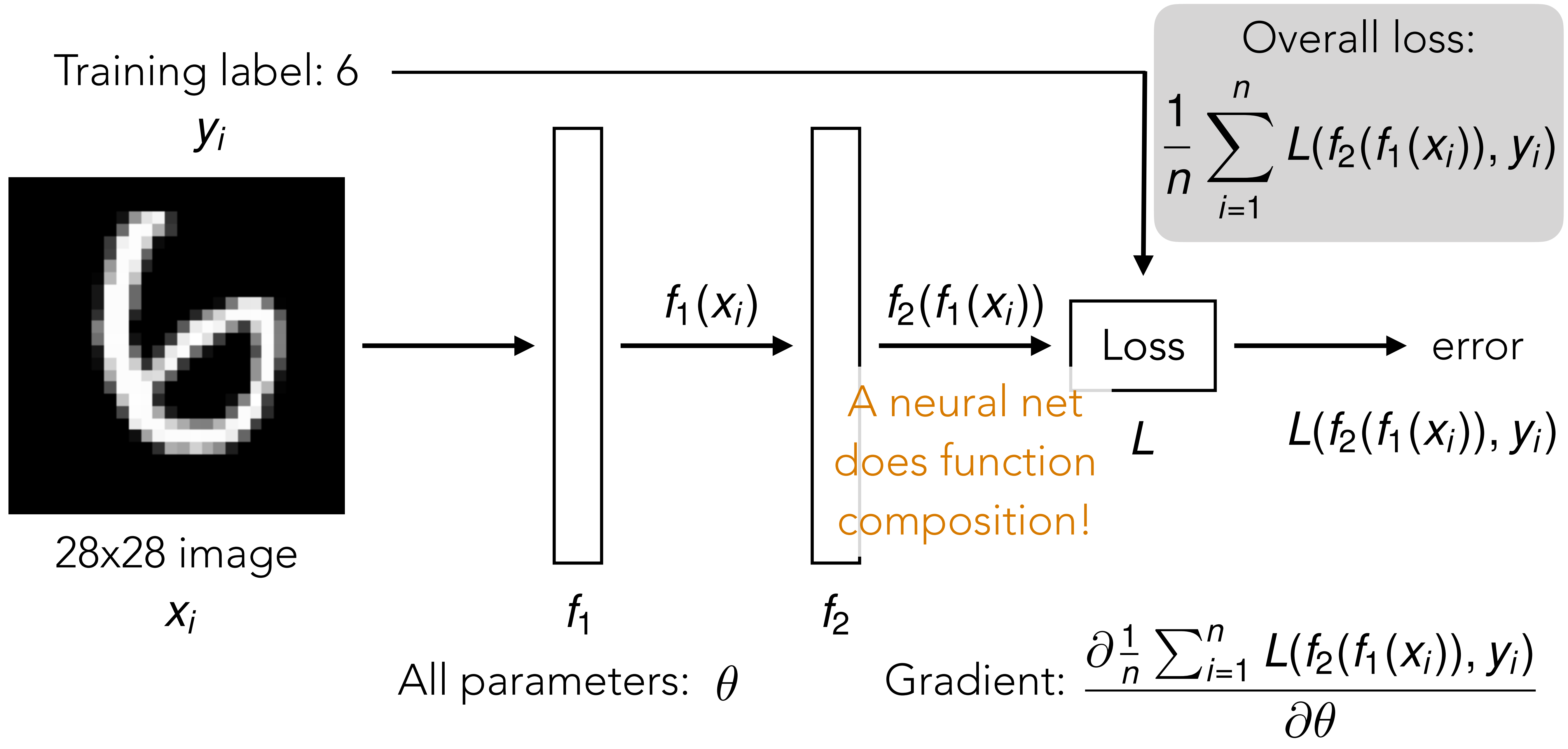
In general: not obvious what error landscape looks like!
➔ we wouldn't know there's a better solution beyond the hill

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

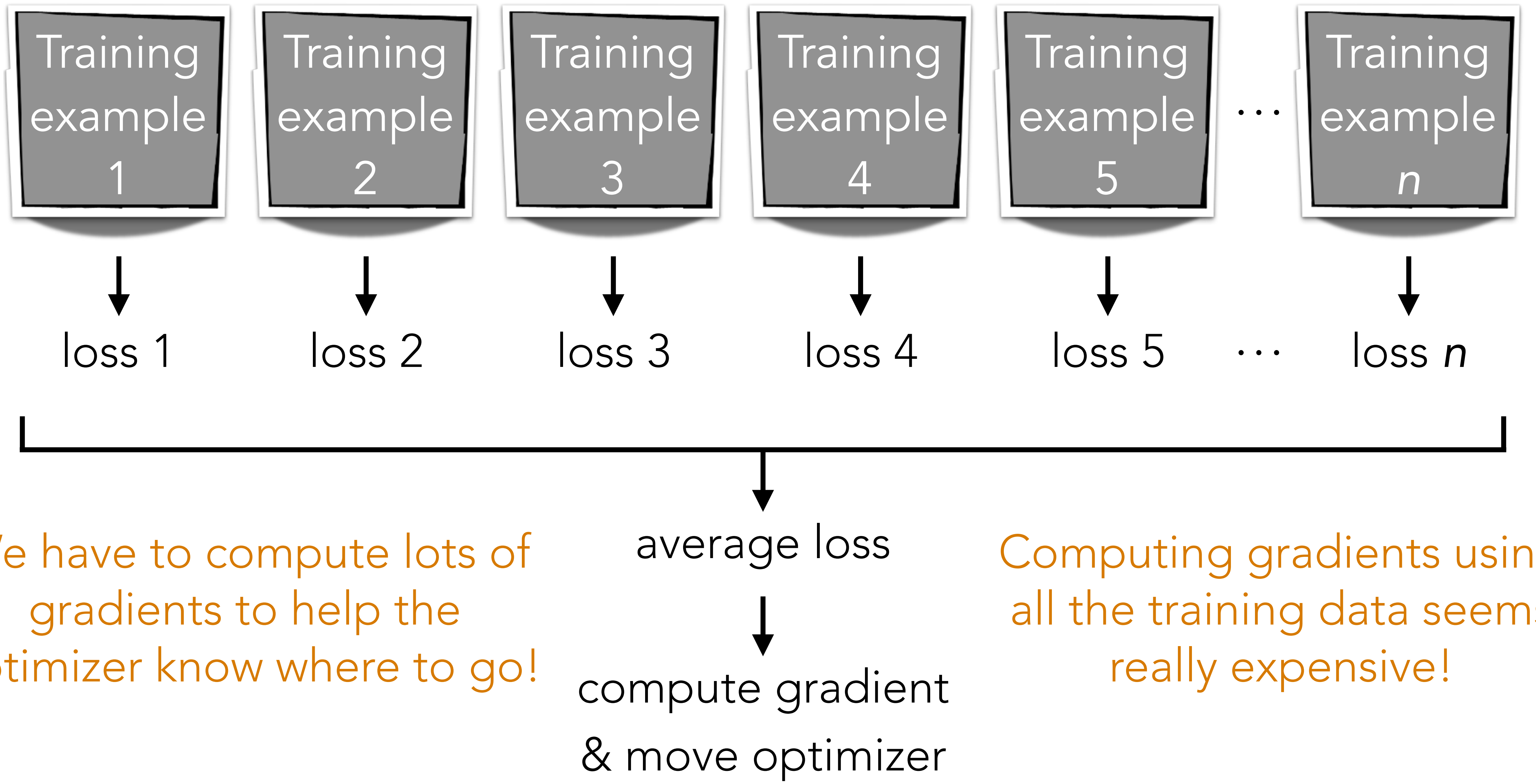
Handwritten Digit Recognition


Training label: 6
y_i
Overall loss:
$\frac{1}{n}\sum_{i=1}^{n} L(f_2(f_1(x_i)), y_i)$
$f_1(x_i)$
$f_2(f_1(x_i))$
Loss
error
A neural net does function composition!
L
$L(f_2(f_1(x_i)), y_i)$
28x28 image
x_i
f_1
f_2
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!

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

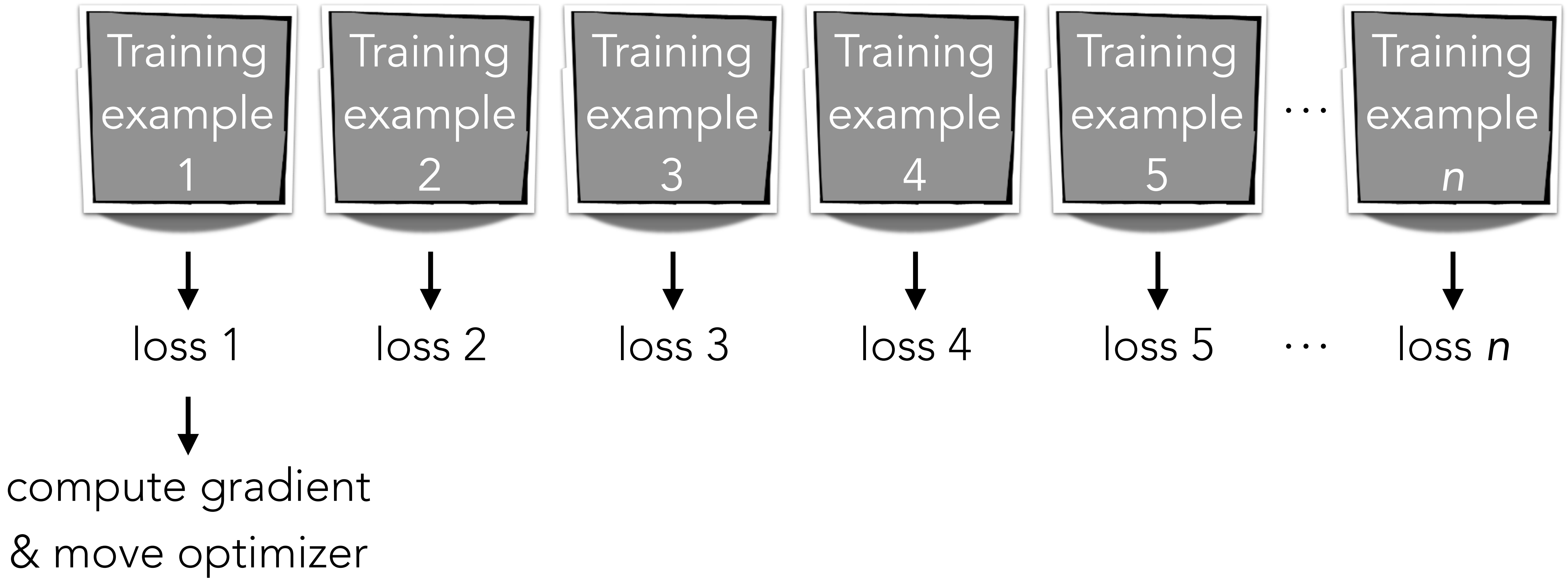
Gradient Descent



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

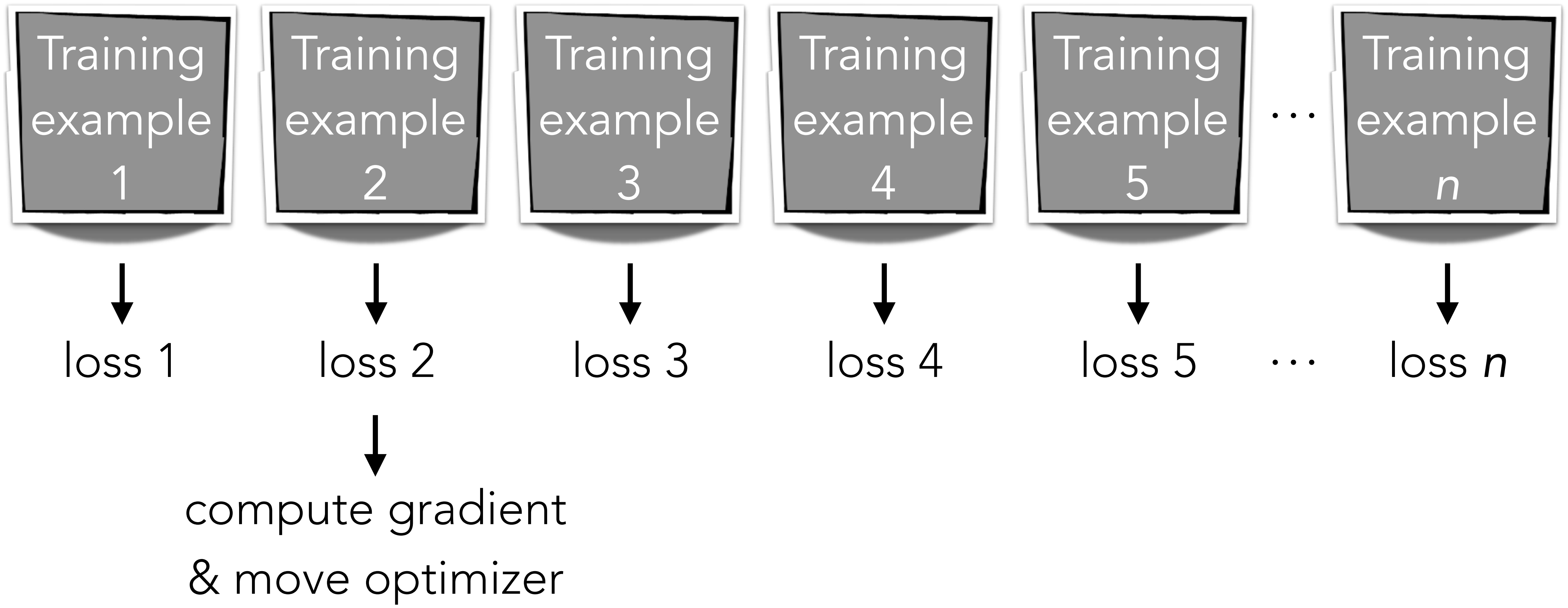
Computing gradients using all the training data seems really expensive!

Stochastic Gradient Descent (SGD)



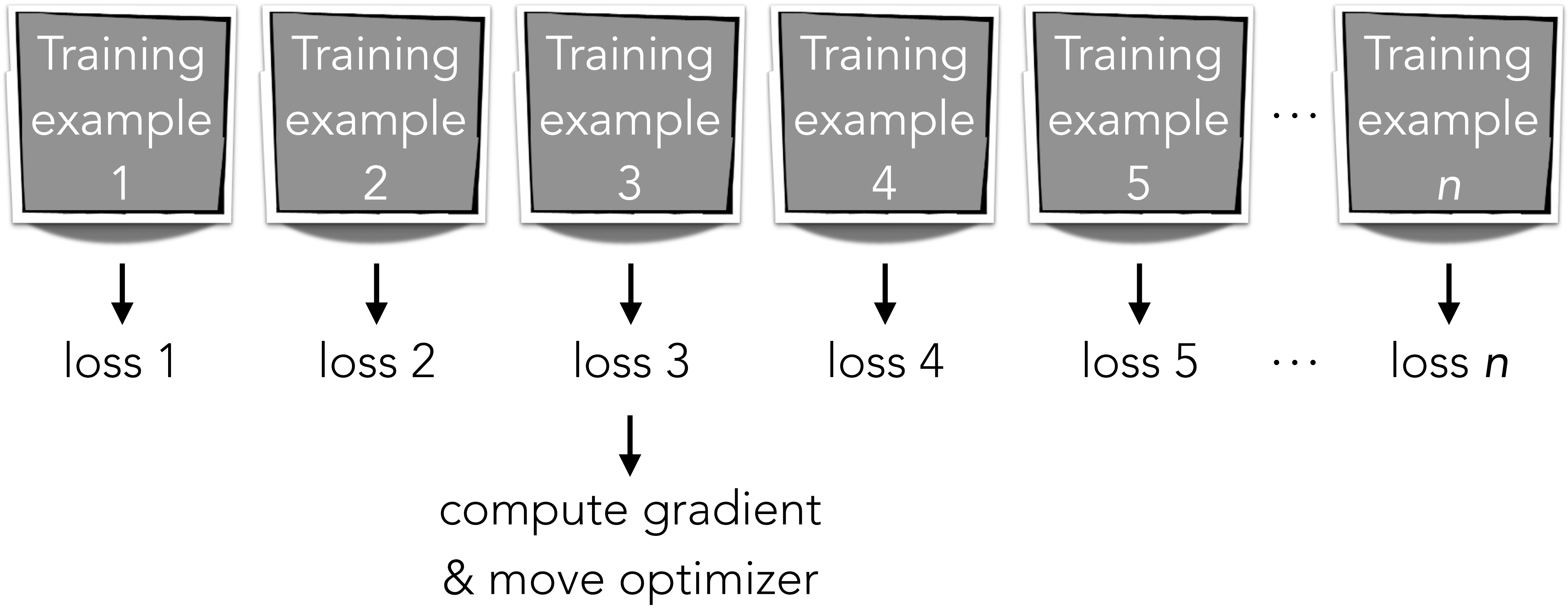
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



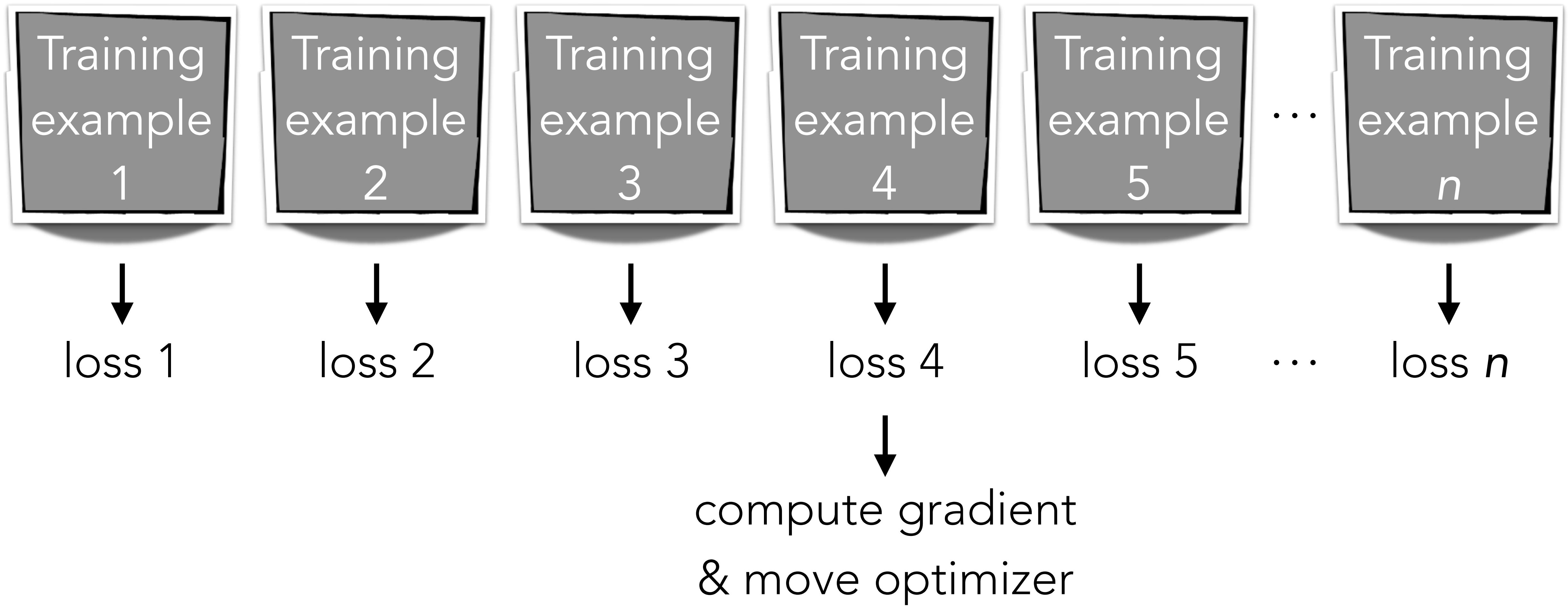
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



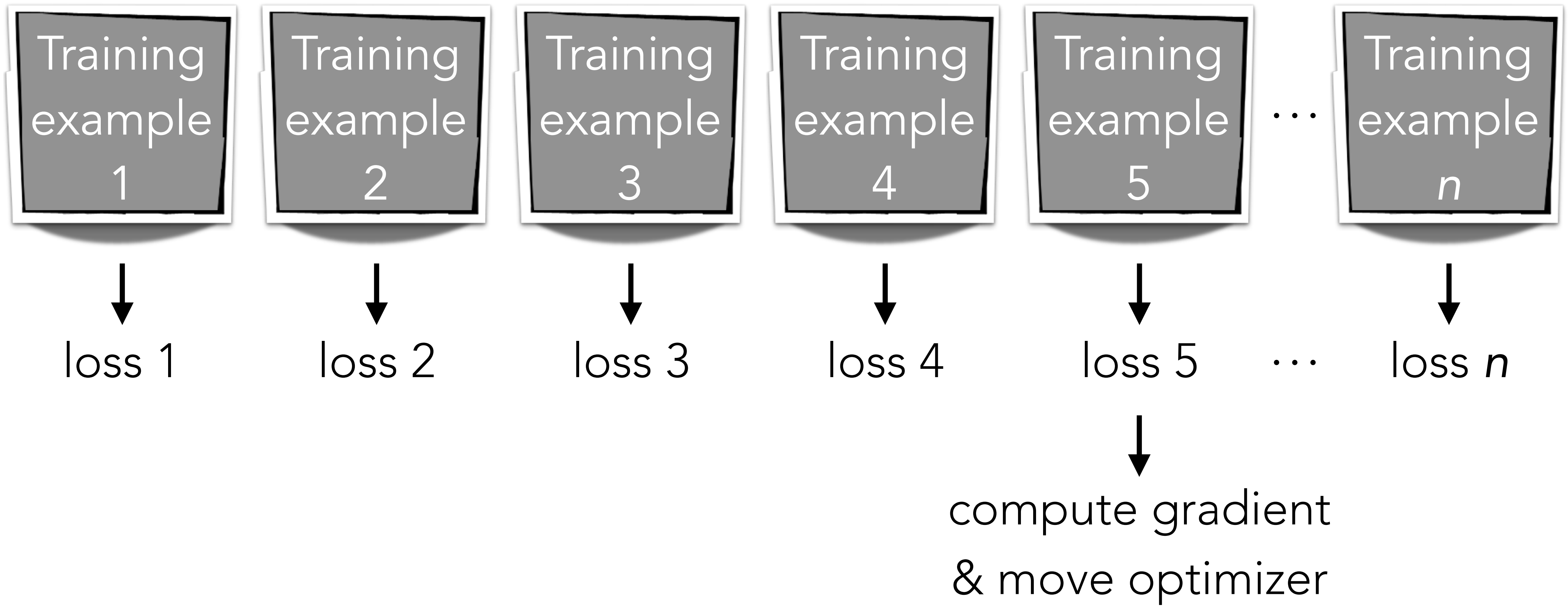
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



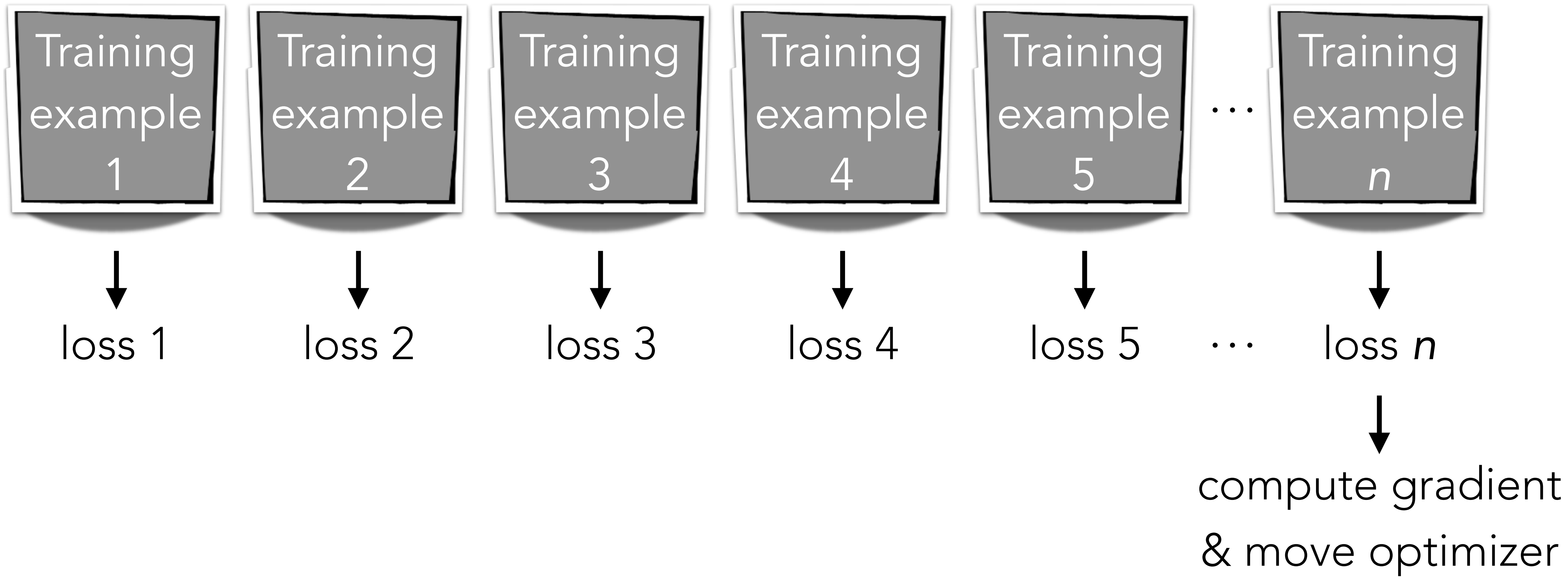
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the “full” gradient)

Stochastic Gradient Descent (SGD)



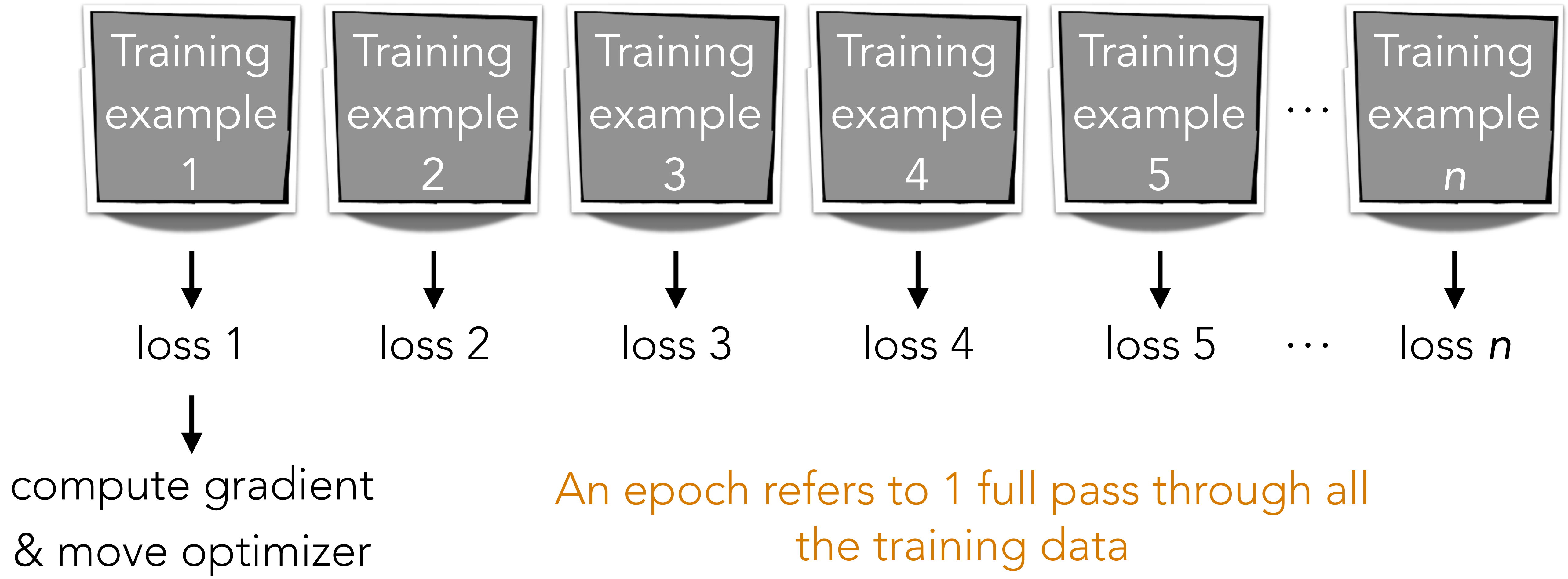
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

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)

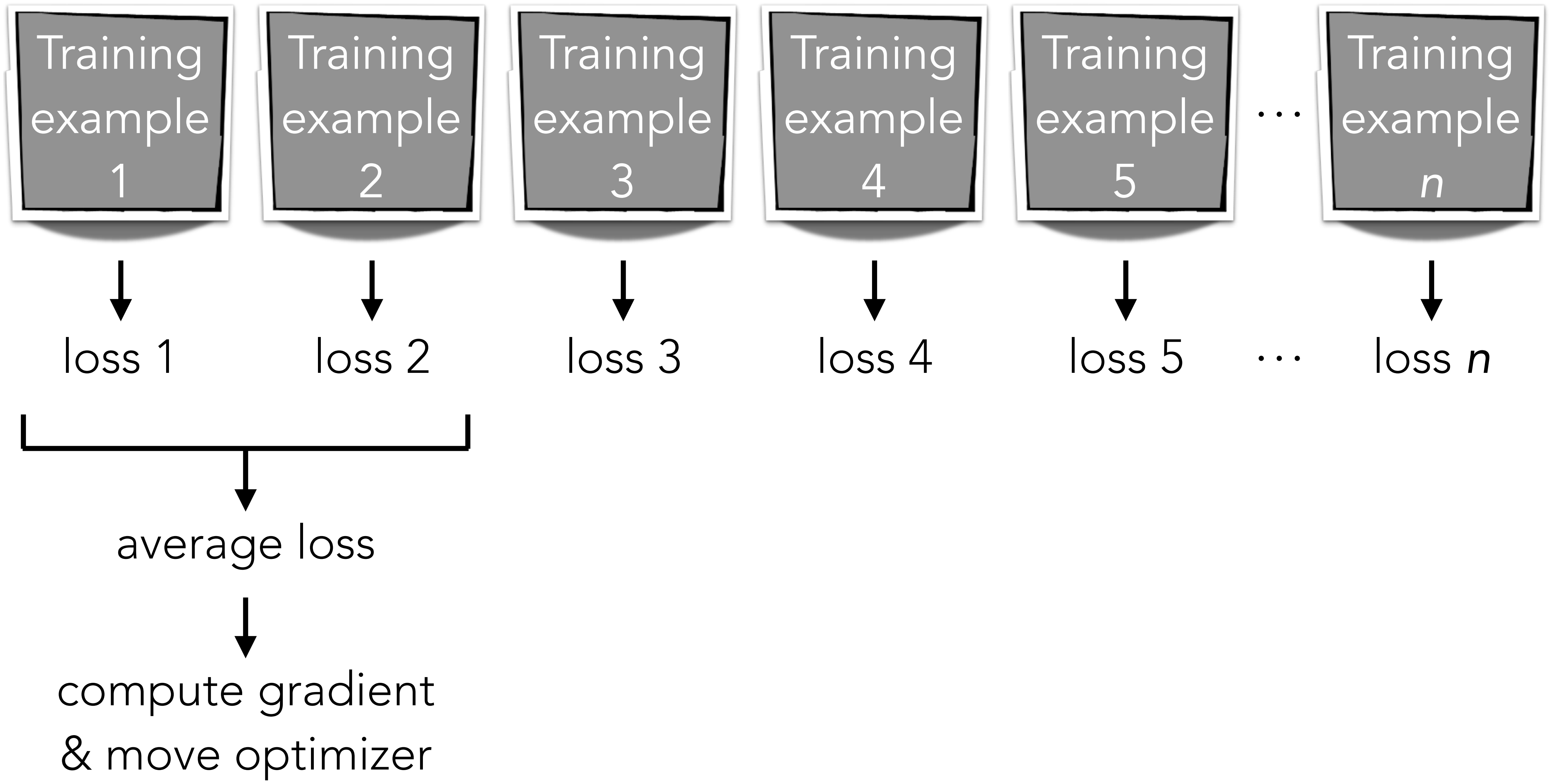
Stochastic Gradient Descent (SGD)



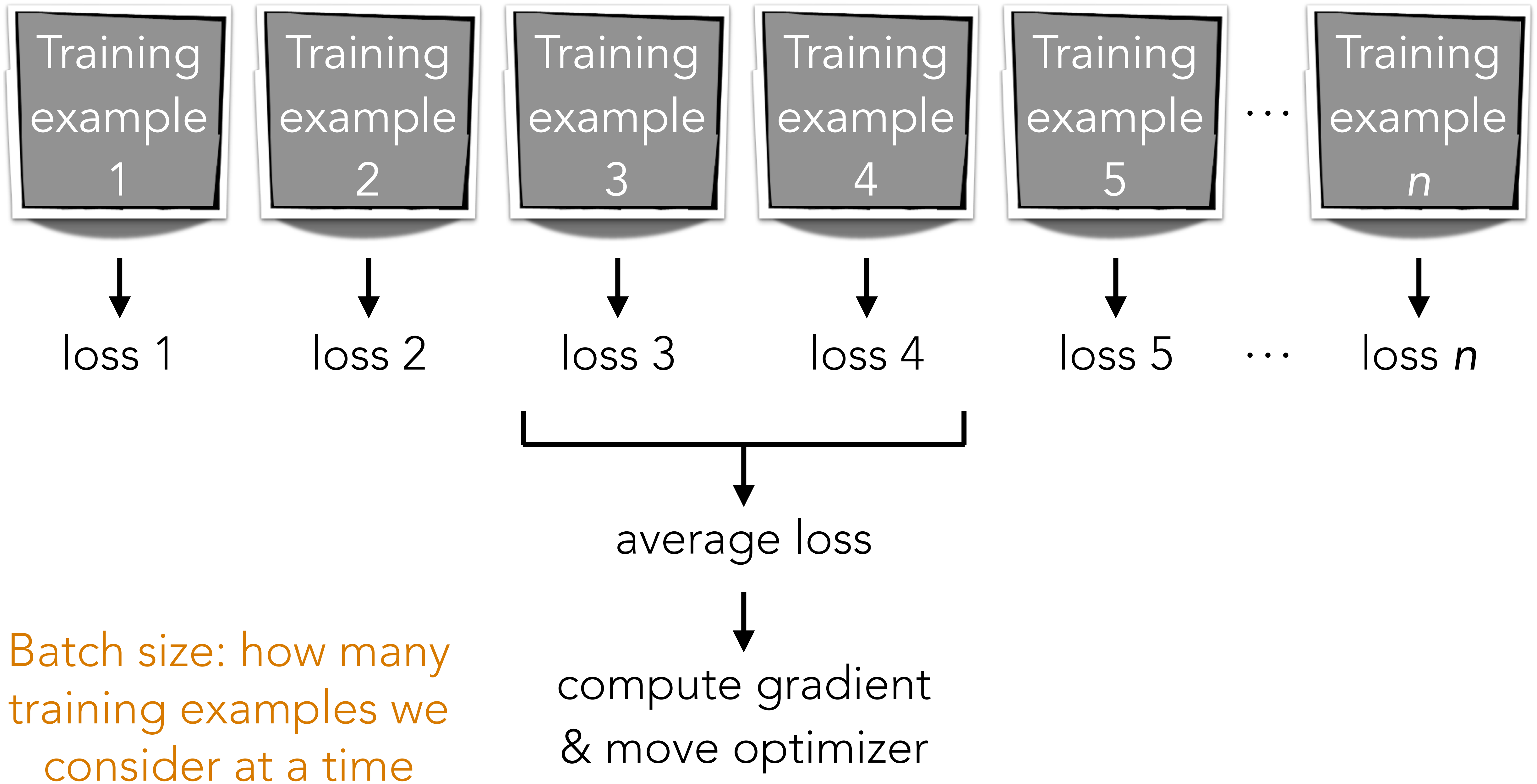
An epoch refers to 1 full pass through all the training data

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



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`

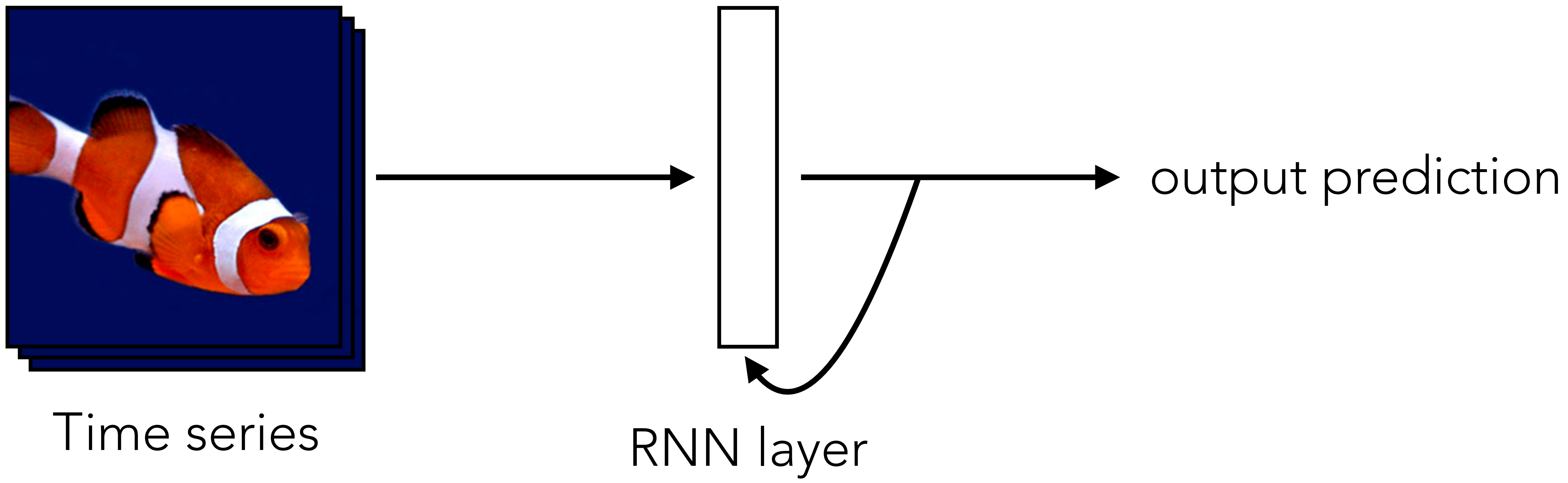
A special kind of RNN: an “LSTM”

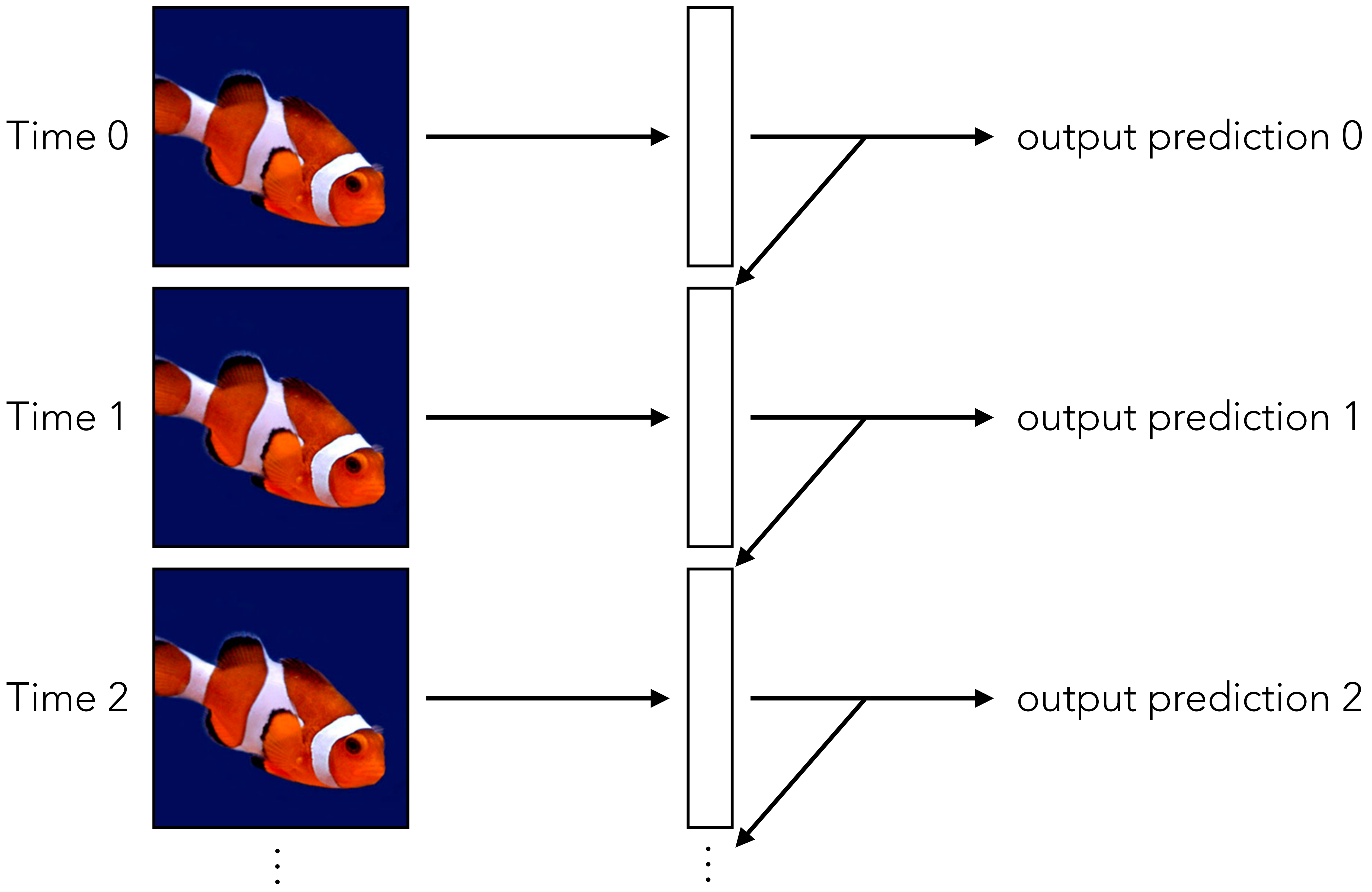
(Flashback) Vanilla ReLU RNN

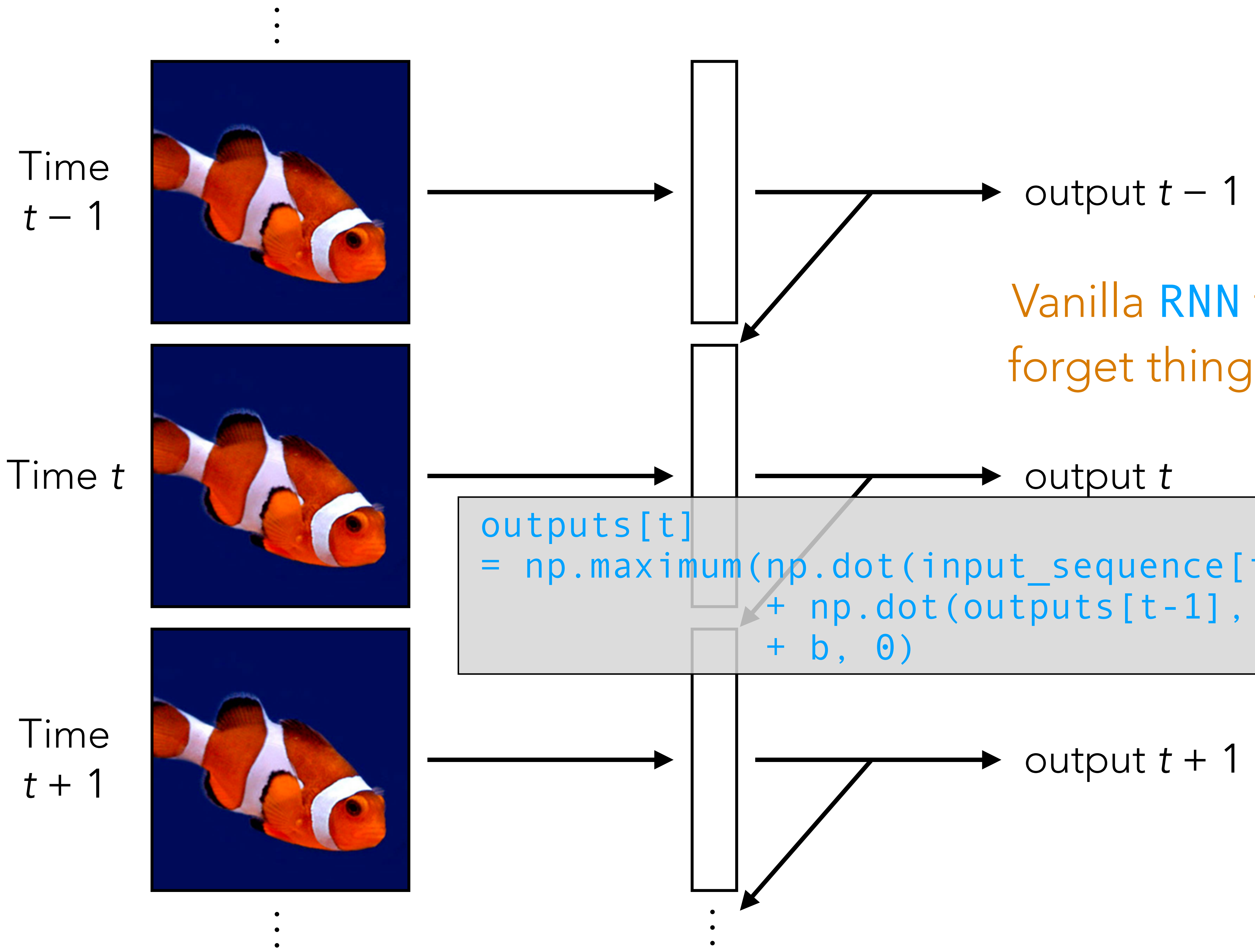
```
current_state = np.zeros(num_nodes)
outputs = []
for input in input_sequence:
  linear = np.dot(input, W.T) + b   \
           + np.dot(current_state, U.T)
  output = np.maximum(0, linear) # ReLU
  outputs.append(output)
  current_state = output
```

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

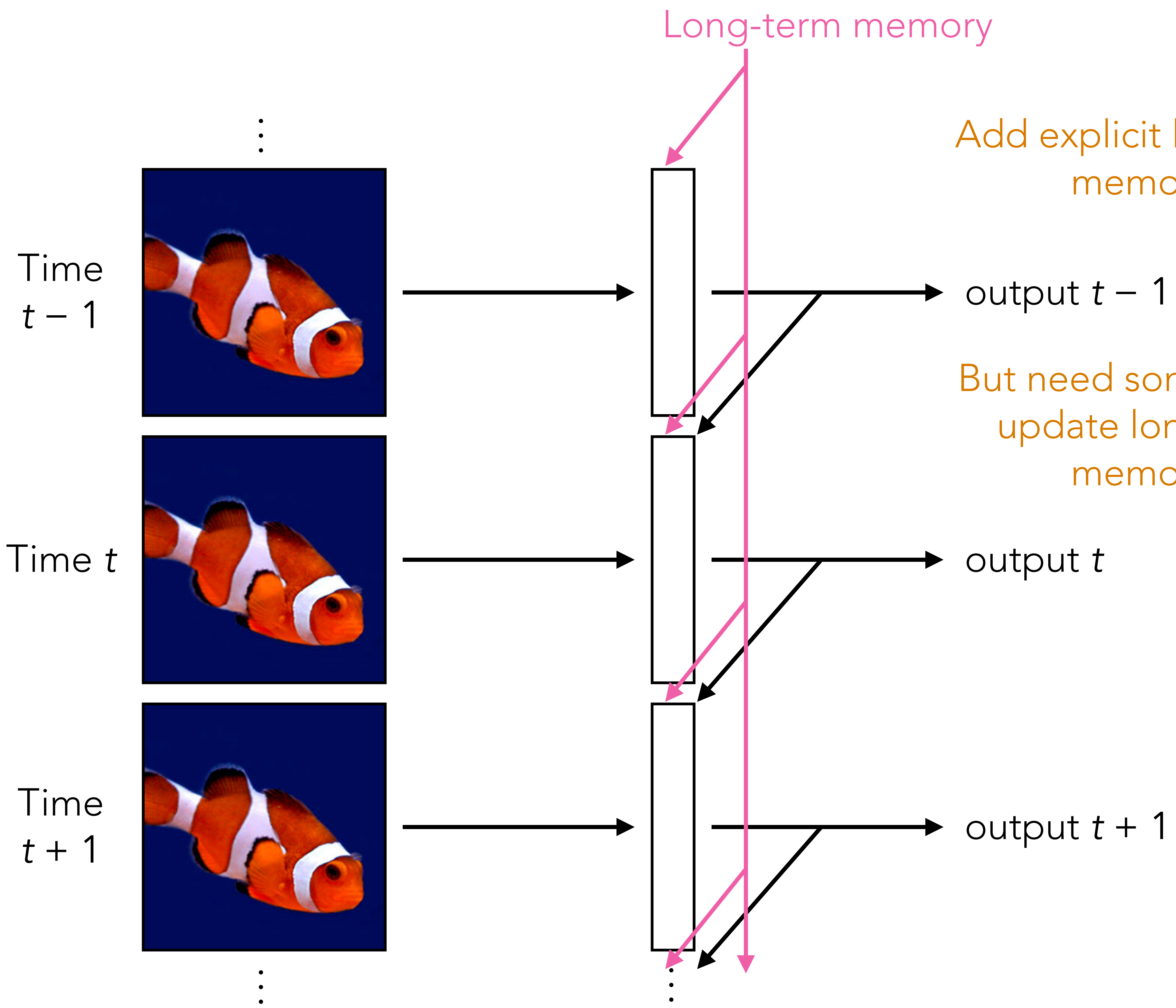

Time series
RNN layer
output prediction


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



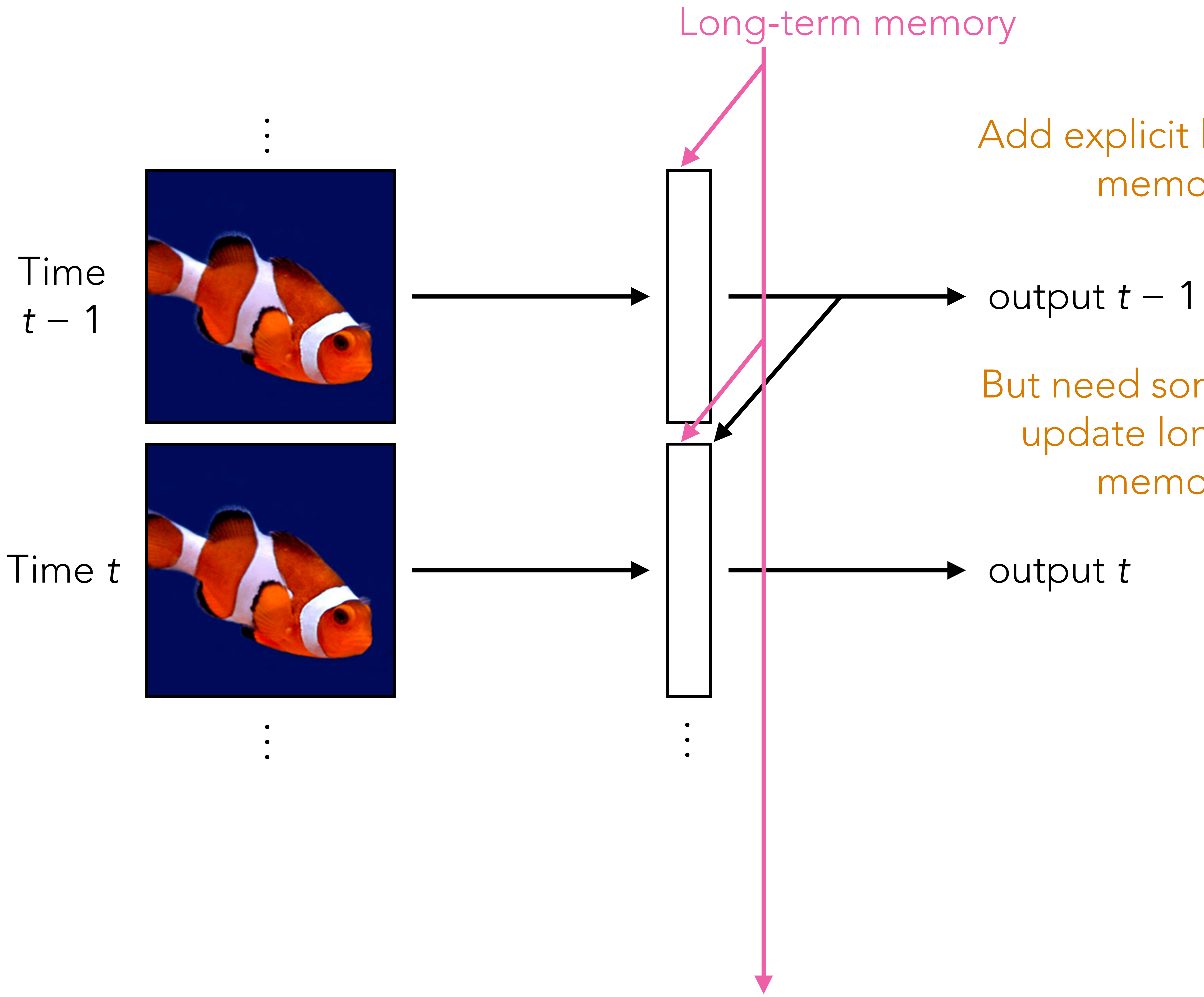
Vanilla RNN tends to forget things quickly

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



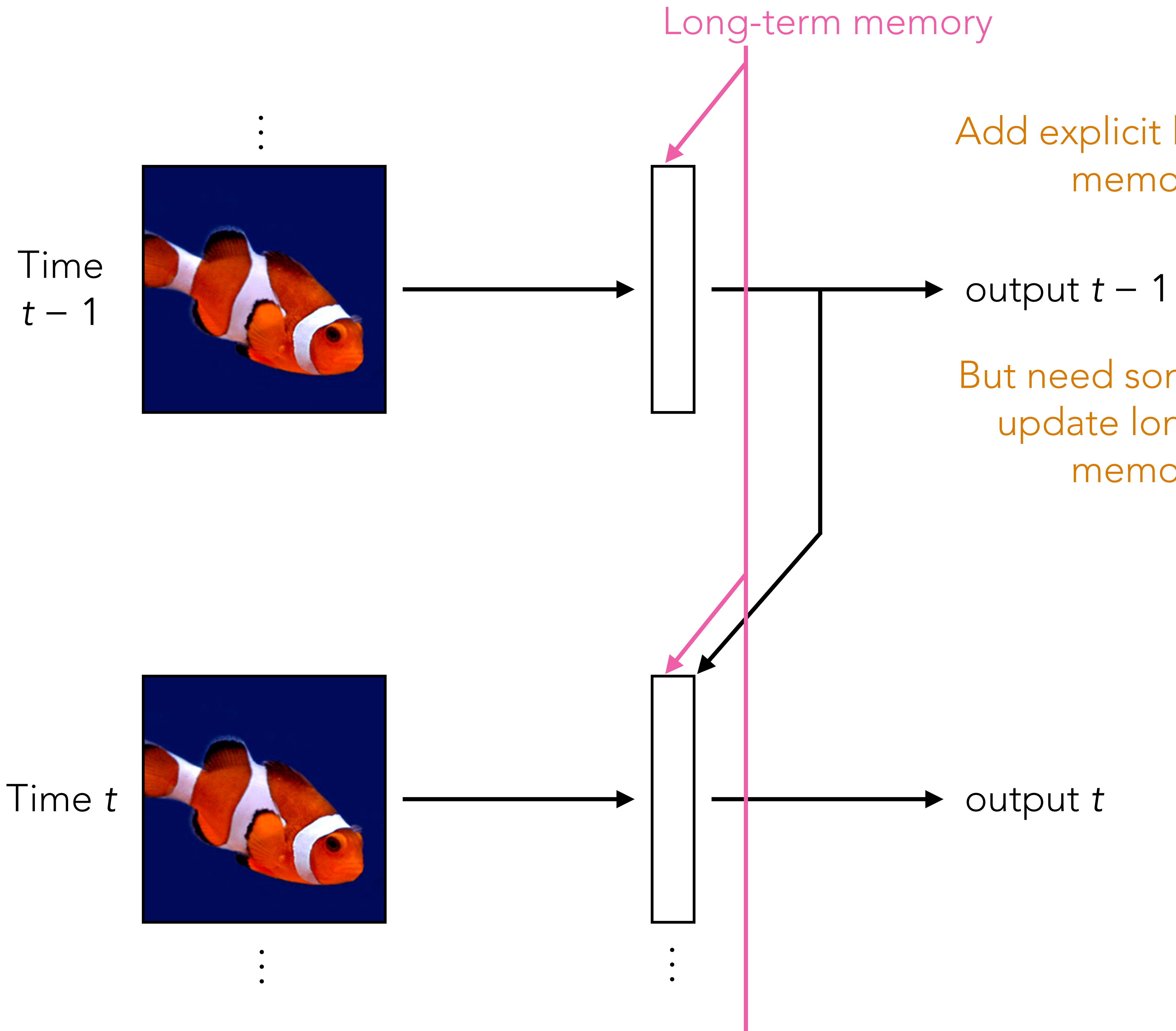
Add explicit long-term memory!

But need some way to update long-term memory!



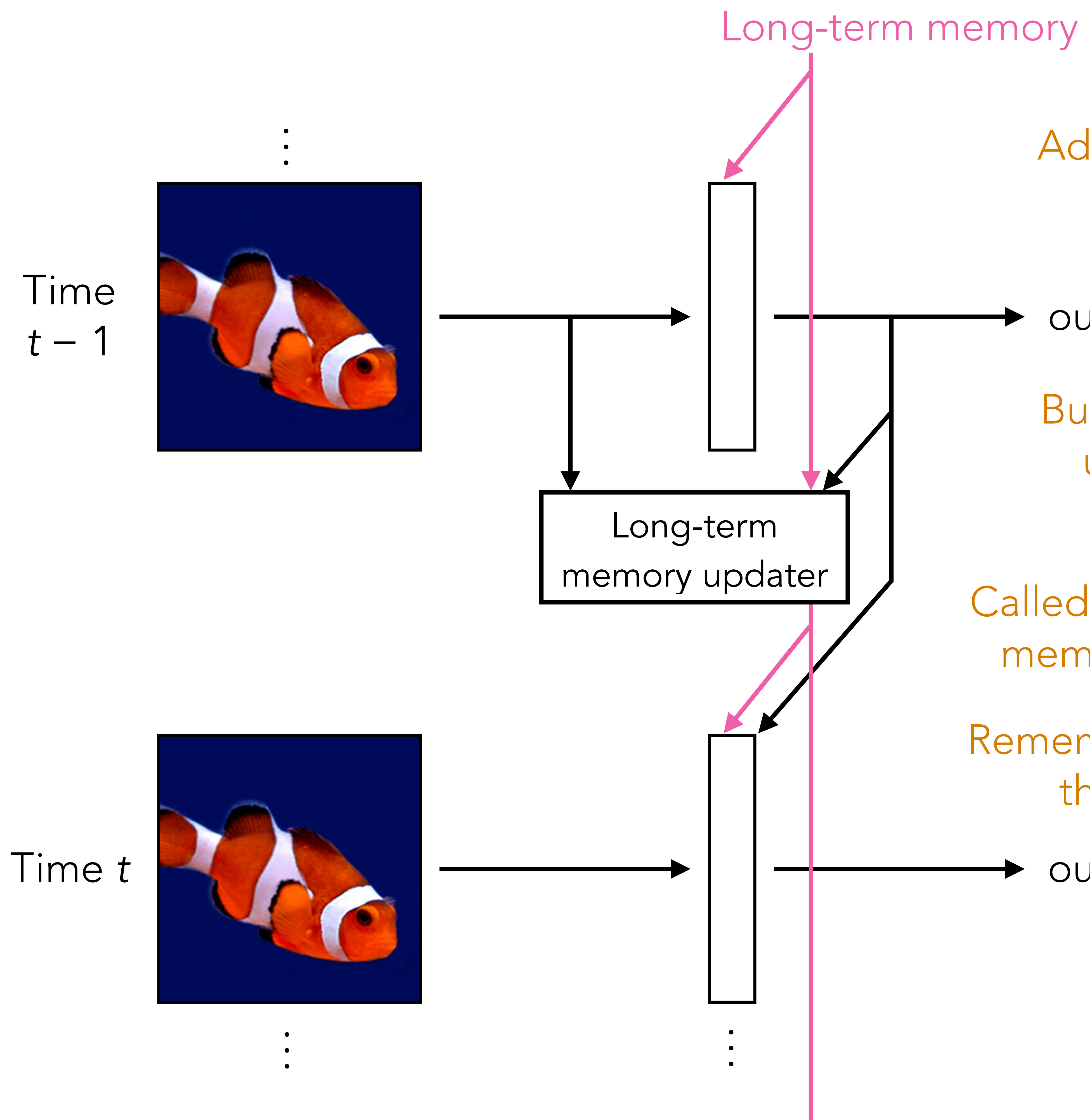
Add explicit long-term memory!

But need some way to update long-term memory!



Add explicit long-term memory!

But need some way to update long-term memory!



Add explicit long-term memory!

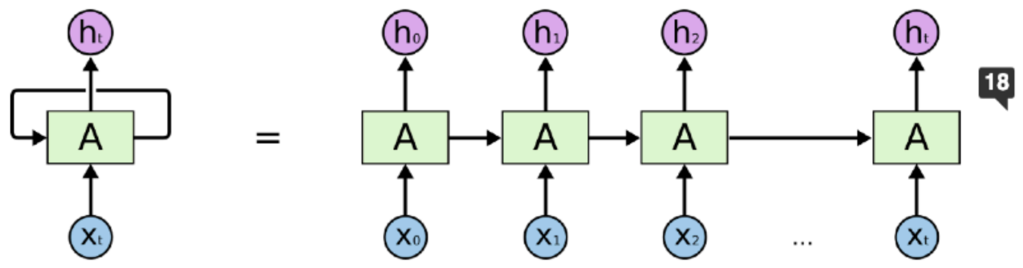
But need some way to update long-term memory!

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

Remembers things longer than vanilla `RNN`

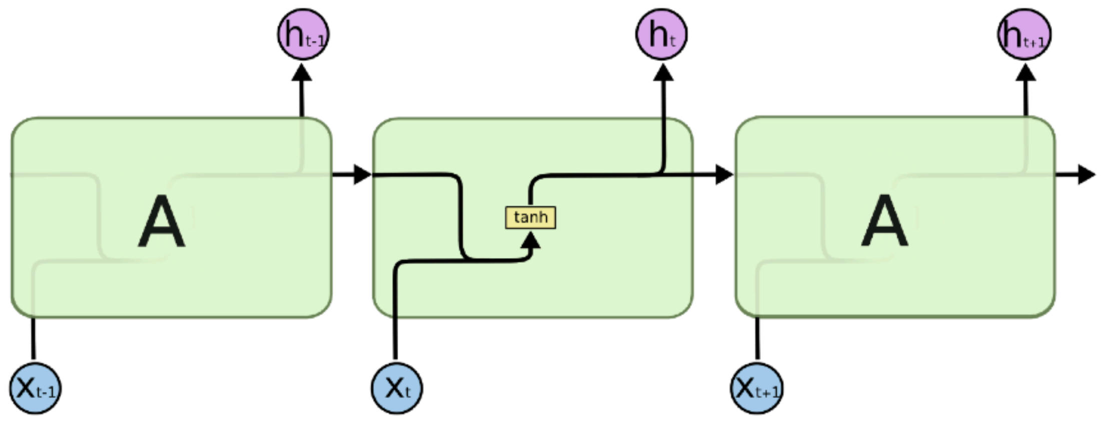
LSTM Parameters

We already saw how to unroll an RNN

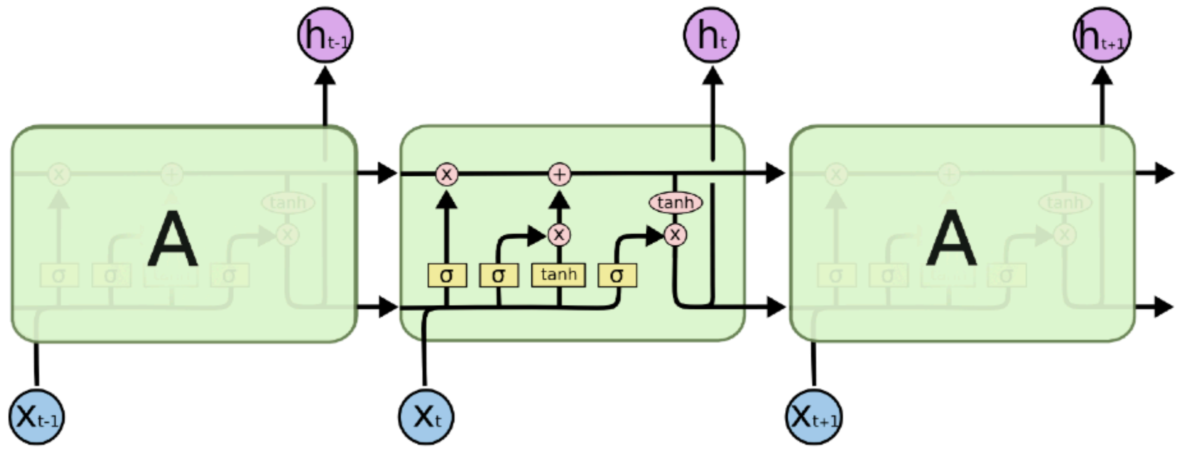


An unrolled recurrent neural network.

Reasoned about the problem of long-term dependencies and LSTM's solution



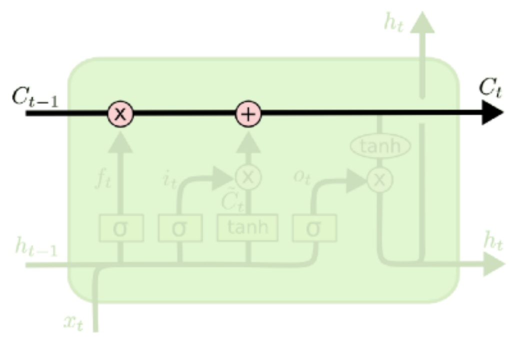
The repeating module in a standard RNN contains a single layer.



The repeating module in an LSTM contains four interacting layers.

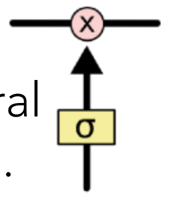
Let us look a bit more into the detail of LSTM and the parameters

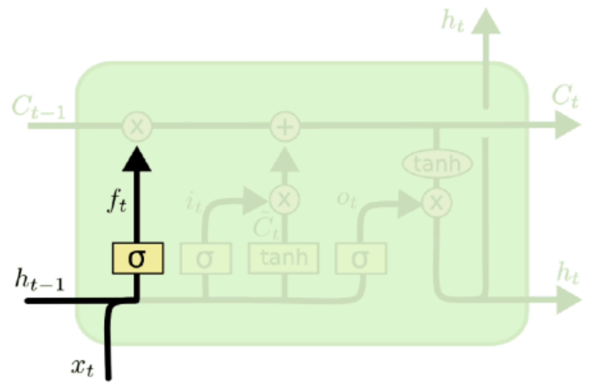
LSTM Parameters



LSTM has 3 gates to protect the cell state

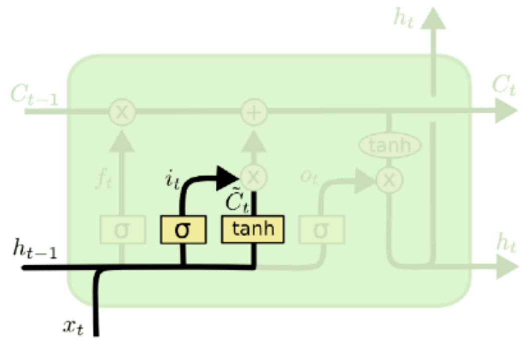
Gates are a way to optionally let information through. They are composed out of a sigmoid neural net layer and a pointwise multiplication operation.





$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

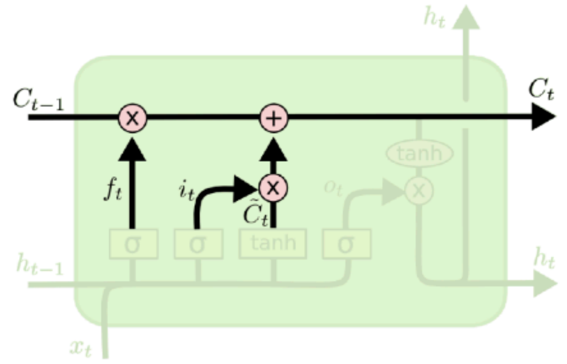
Forget the old information because of the new relevant information coming in



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

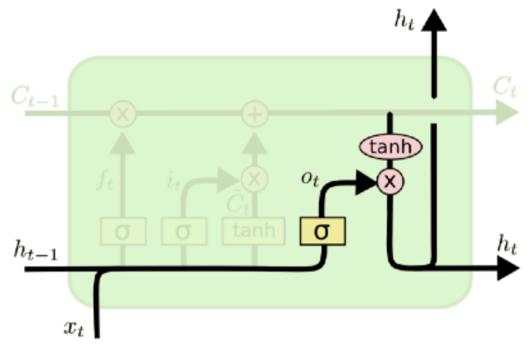
Decide what new information we're going to store in the cell state:

1. "Input gate" layer for which values to update.
2. Tanh layer creating new candidate values



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Update the old cell state
Multiply the old state by forgetting the things we decided to forget earlier.
Then we add the new candidate values, scaled by how much we decided to update each state value.



$$o_t = \sigma\left(W_o \ [h_{t-1}, x_t] \ + \ b_o\right)$$
$$h_t = o_t * \tanh\left(C_t\right)$$

Output: filtered version of the cell state
Put the cell state through tanh (to push the values to be between −1 and 1) and multiply it by the output of the sigmoid gate, so that we only output the parts we decided to.

UDA_pytorch_utils.py

Let's look at `UDA_get_rnn_last_time_step_outputs`

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

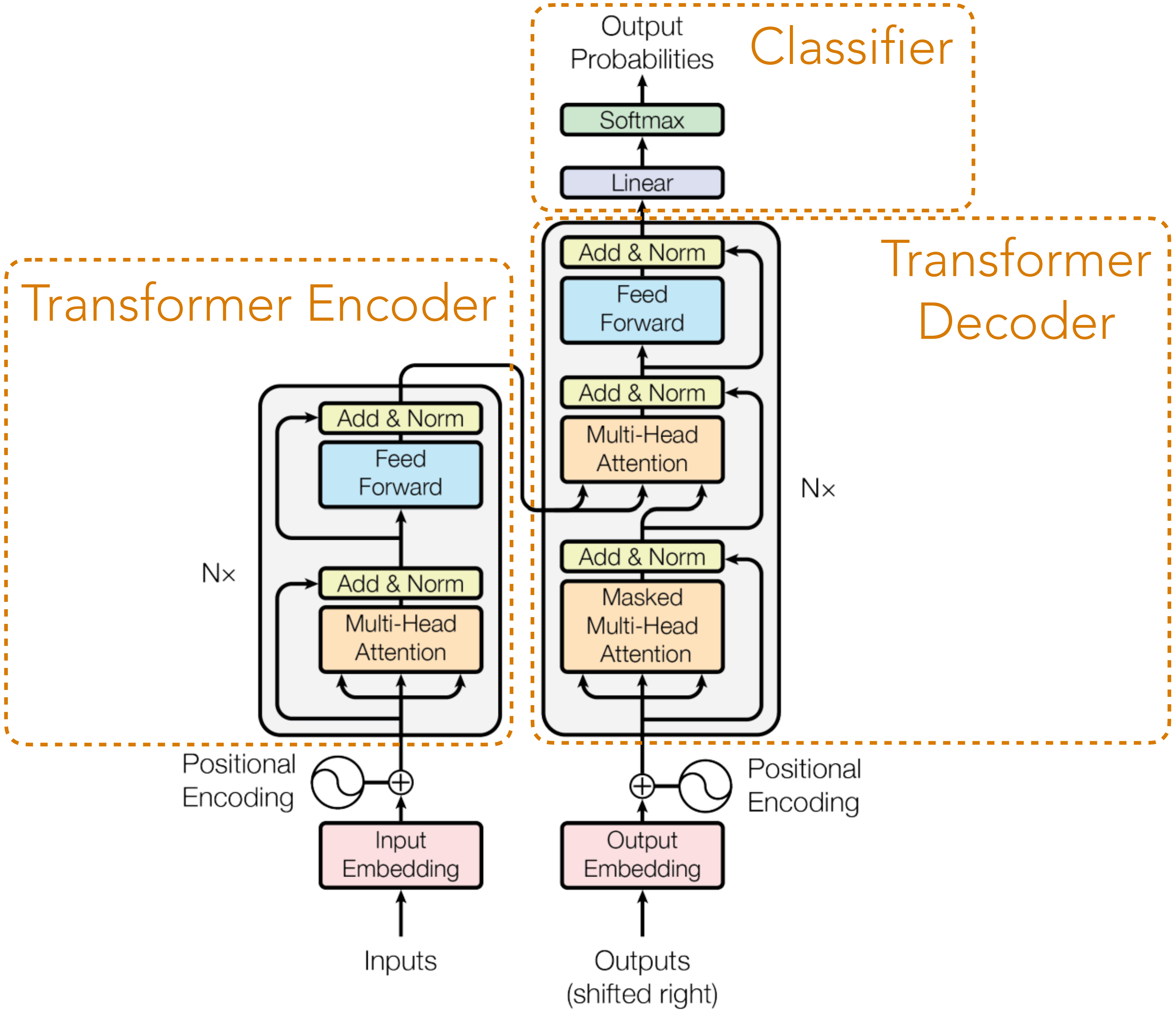


Figure 1: The Transformer - model architecture.

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)

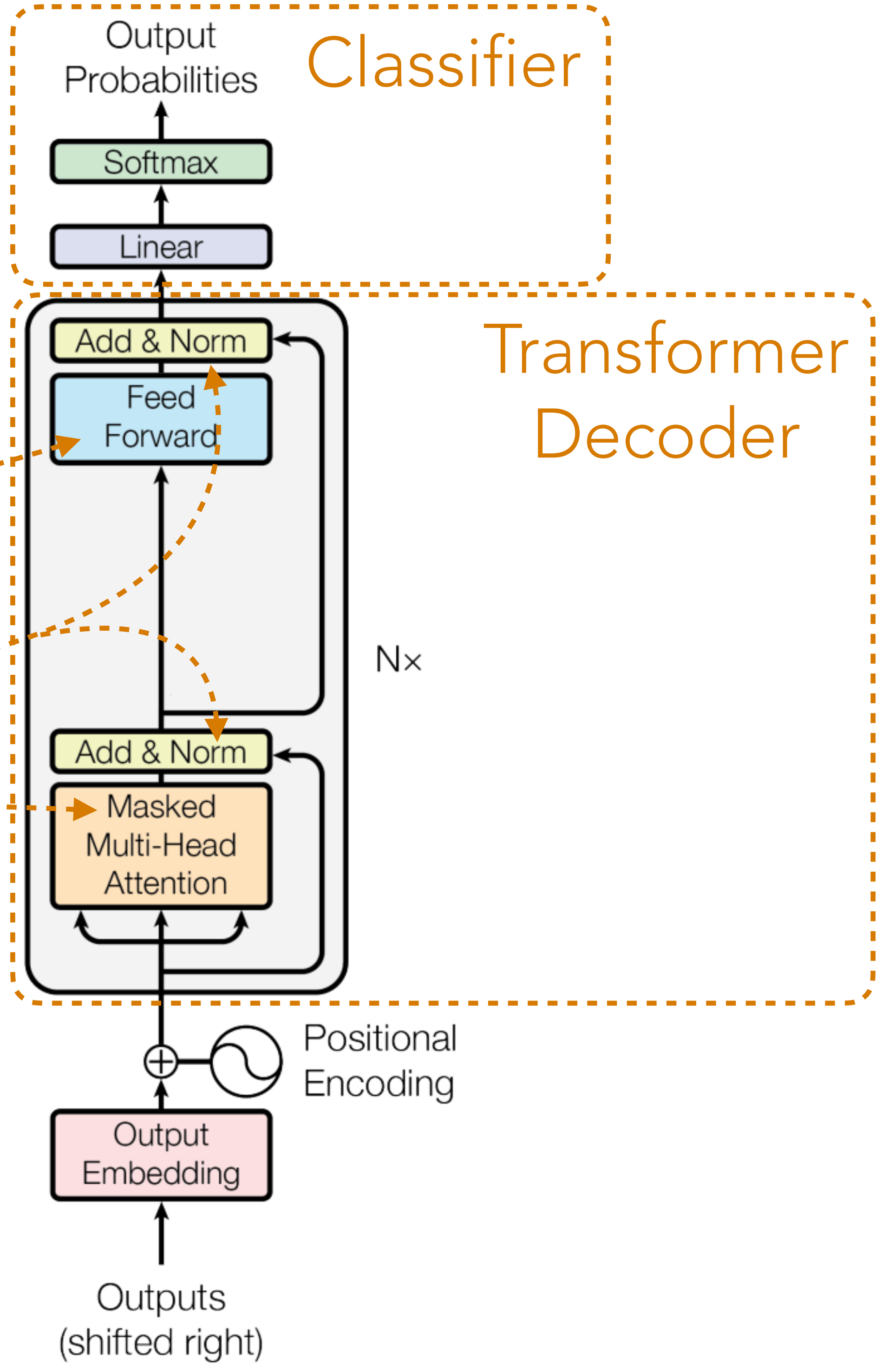
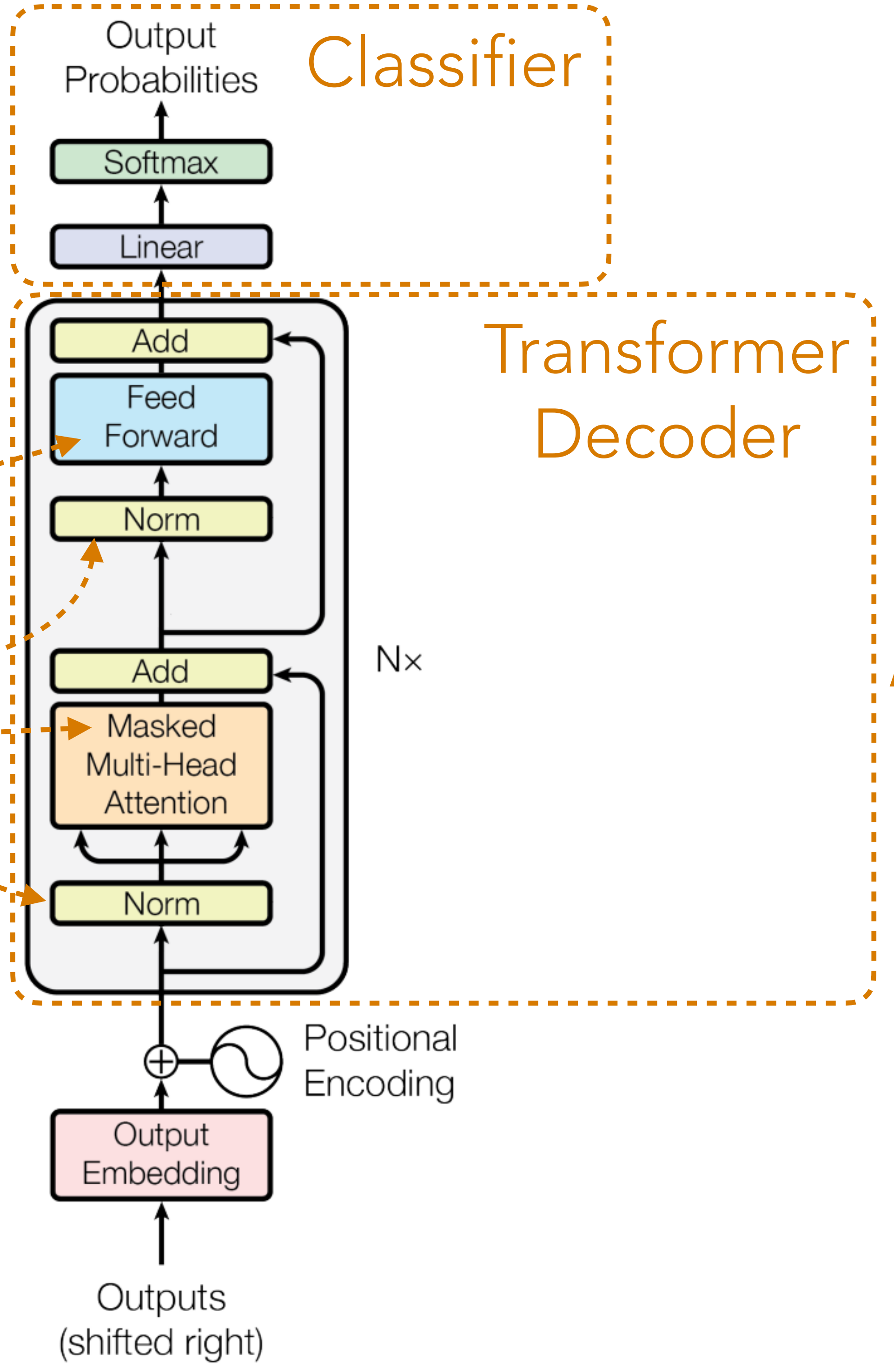


Figure 1: The Transformer - model architecture.

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)

"Pre-norm" version that's now standard

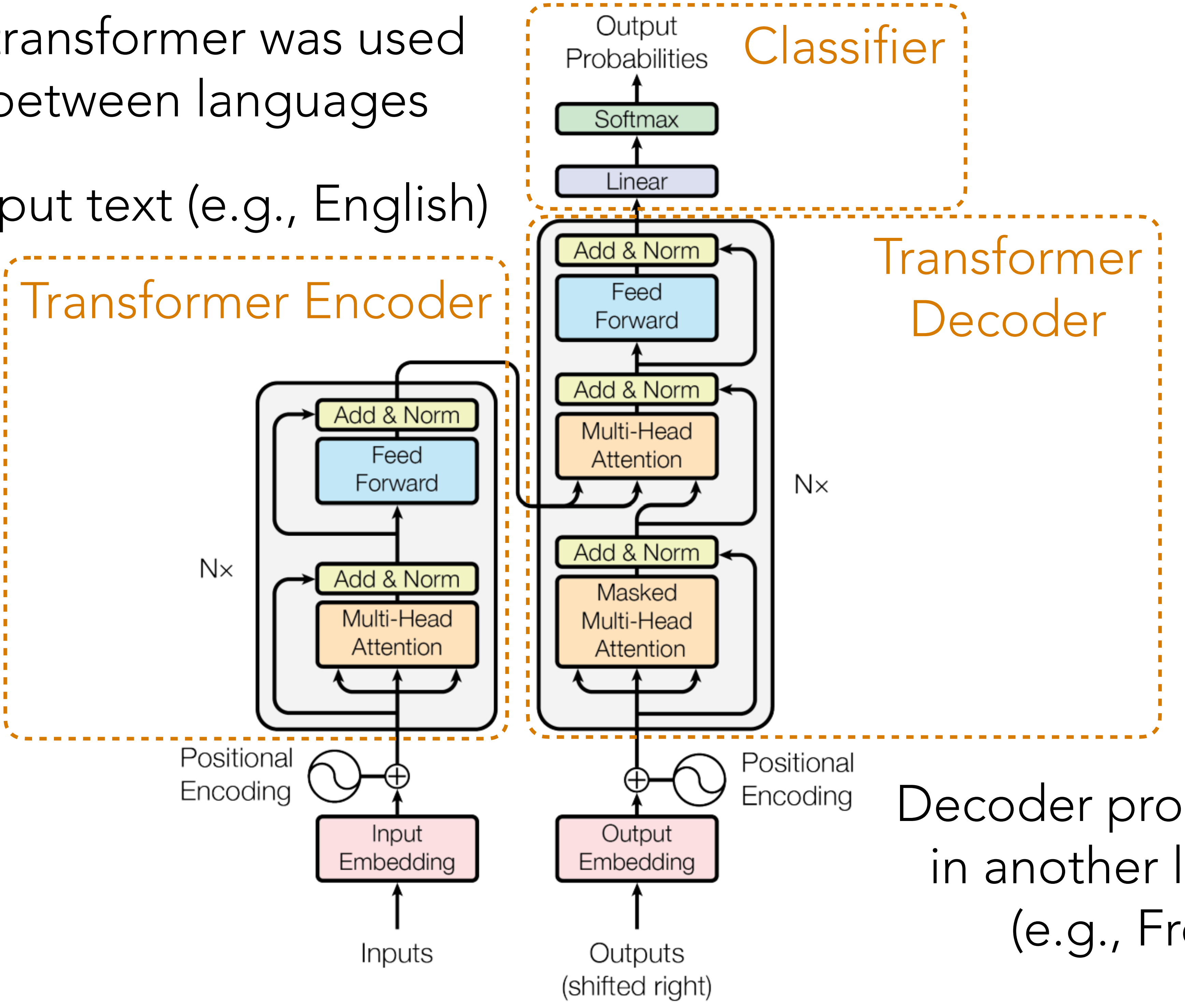
Figure 1: The Transformer - model architecture.

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)



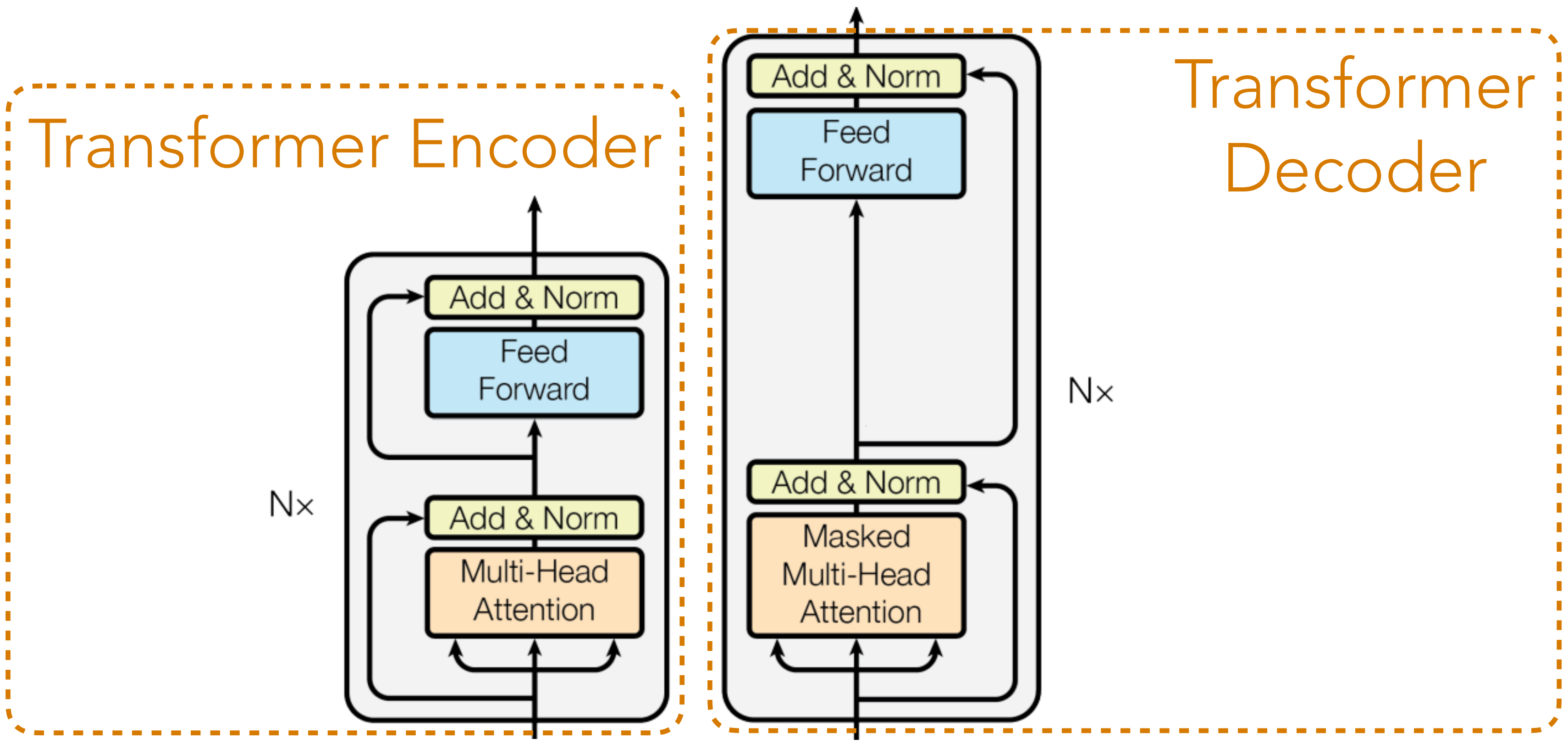
Decoder produces text in another language (e.g., French)

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