Carnegie Mellon University
Heinz College

94-775 Unstructured Data Analytics for Policy

Some slides for part of recitation
(to help with understanding the GPT demo):
constructing a PyTorch model *not* using `nn.Sequential`,
and how training a neural net roughly works

Slides by George H. Chen

(Flashback) Neural Net as Function Approximation

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

Multinomial logistic regression:

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

Today's recitation coverage will reveal where this kind of code shows up in PyTorch

Multilayer perceptron:

```
linear1 = np.dot(input, W1.T) + b1
relu = np.maximum(0, linear1)  # ReLU
linear2 = np.dot(relu, W2.T) + b2
output = softmax(linear2)
```

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

More layers ⇒ computer program has more lines of code

Constructing a neural net in PyTorch using `nn.Module` (instead of `nn.Sequential`)

(this level of detail is needed for understanding the GPT demo)

Constructing PyTorch Models with `nn.Module`

What you've already seen previously:

```
deeper_model = nn.Sequential(nn.Flatten(),
                             nn.Linear(in_features=784, out_features=512),
                             nn.ReLU(),
                             nn.Linear(in_features=512, out_features=10))
```

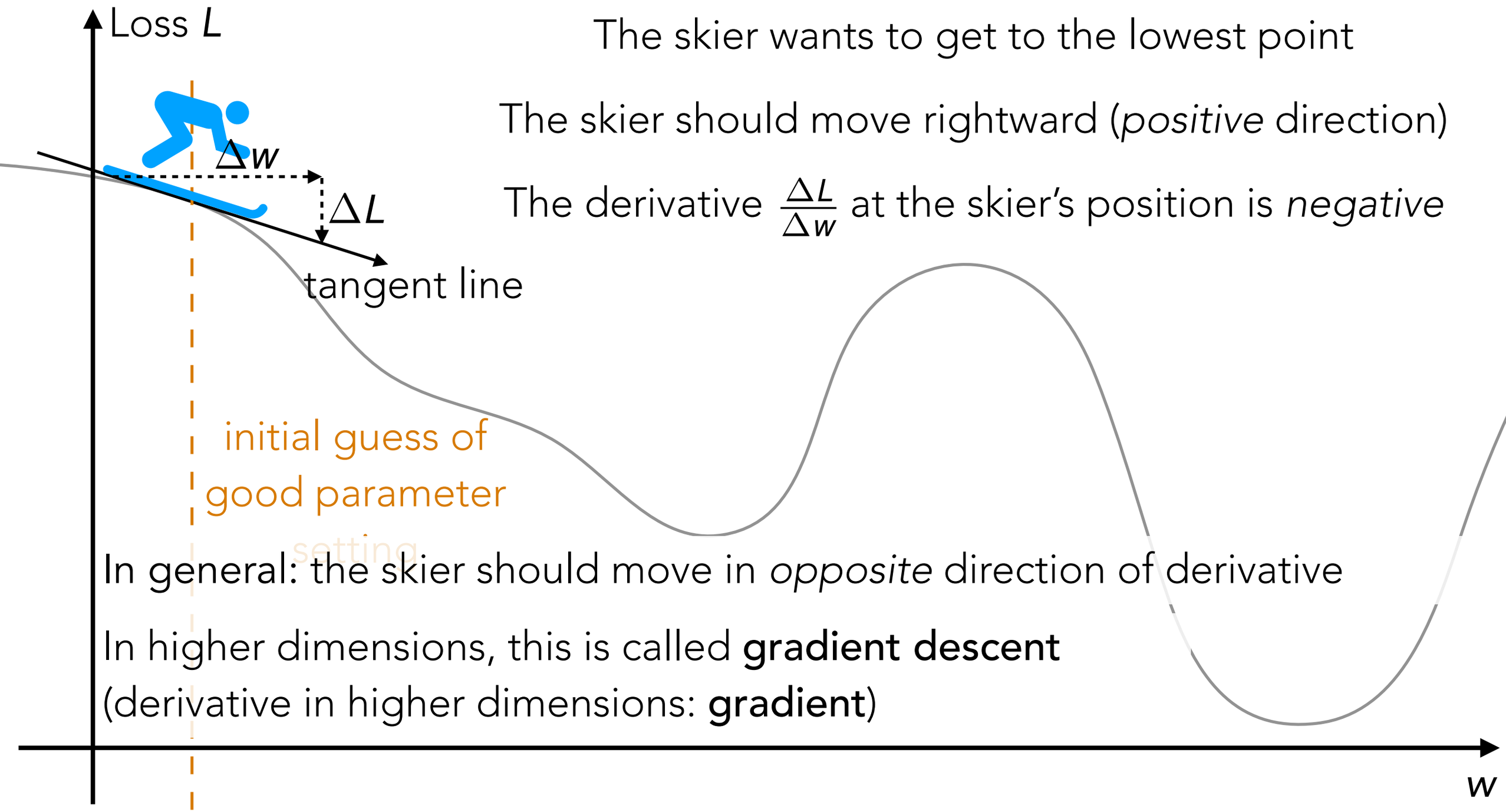
Another way to write this:

```
class DeeperModel(nn.Module):
    def __init__(self, num_in_features, num_intermediate_features, num_out_features):
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear1 = nn.Linear(num_in_features, num_intermediate_features)
        self.relu = nn.ReLU()
        self.linear2 = nn.Linear(num_intermediate_features, num_out_features)

    def forward(self, inputs):
        flatten_output = self.flatten(inputs)
        linear1_output = self.linear1(flatten_output)
        relu_output = self.relu(linear1_output)
        linear2_output = self.linear2(relu_output)
        return linear2_output

deeper_model = DeeperModel(784, 512, 10)
```

This is like the NumPy code presented earlier for the multilayer perceptron (softmax is excluded as it's part of the loss calculation)

Training 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)

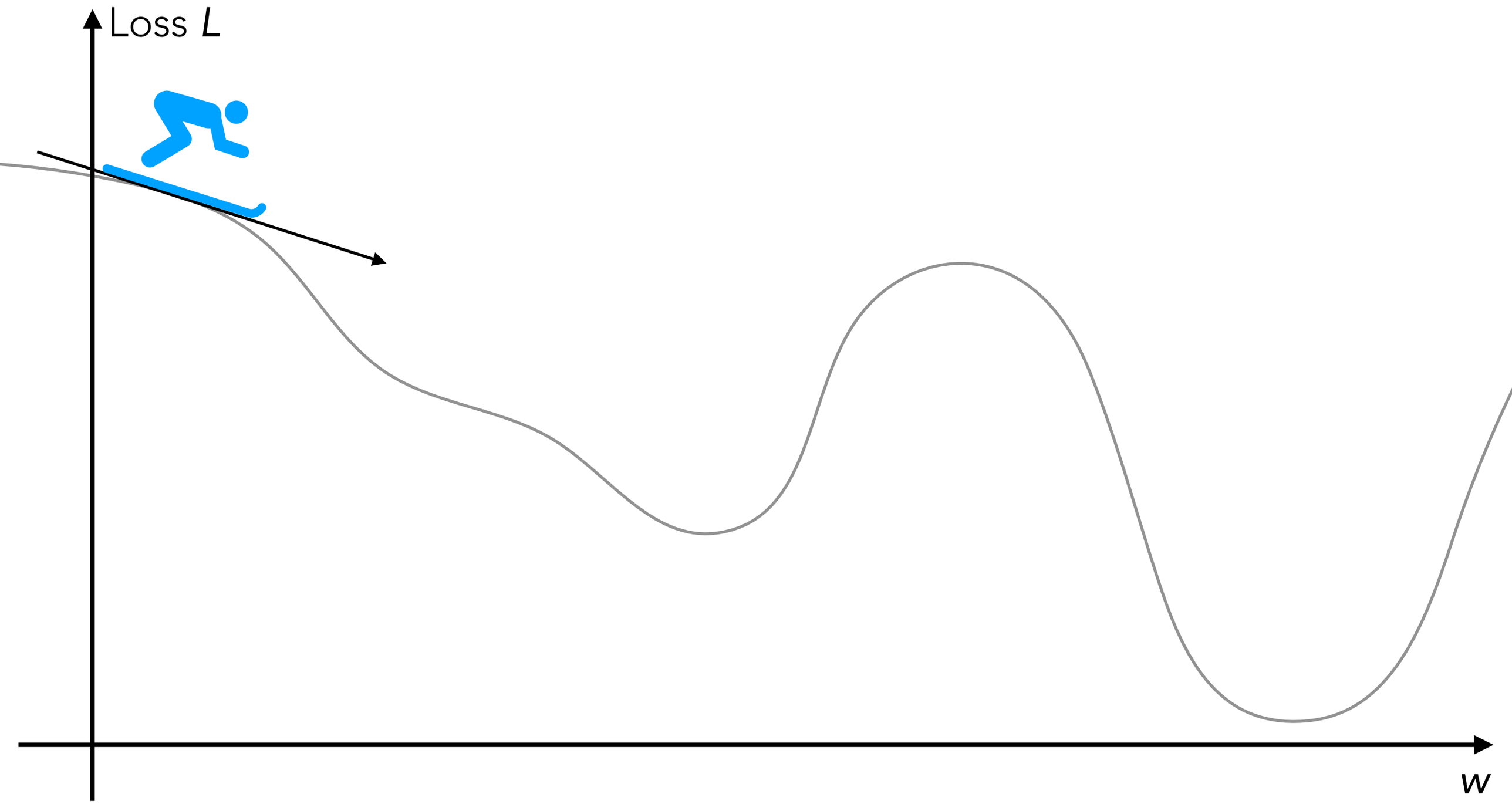
The derivative $\frac{\Delta L}{\Delta w}$ at the skier's position is *negative*

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

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

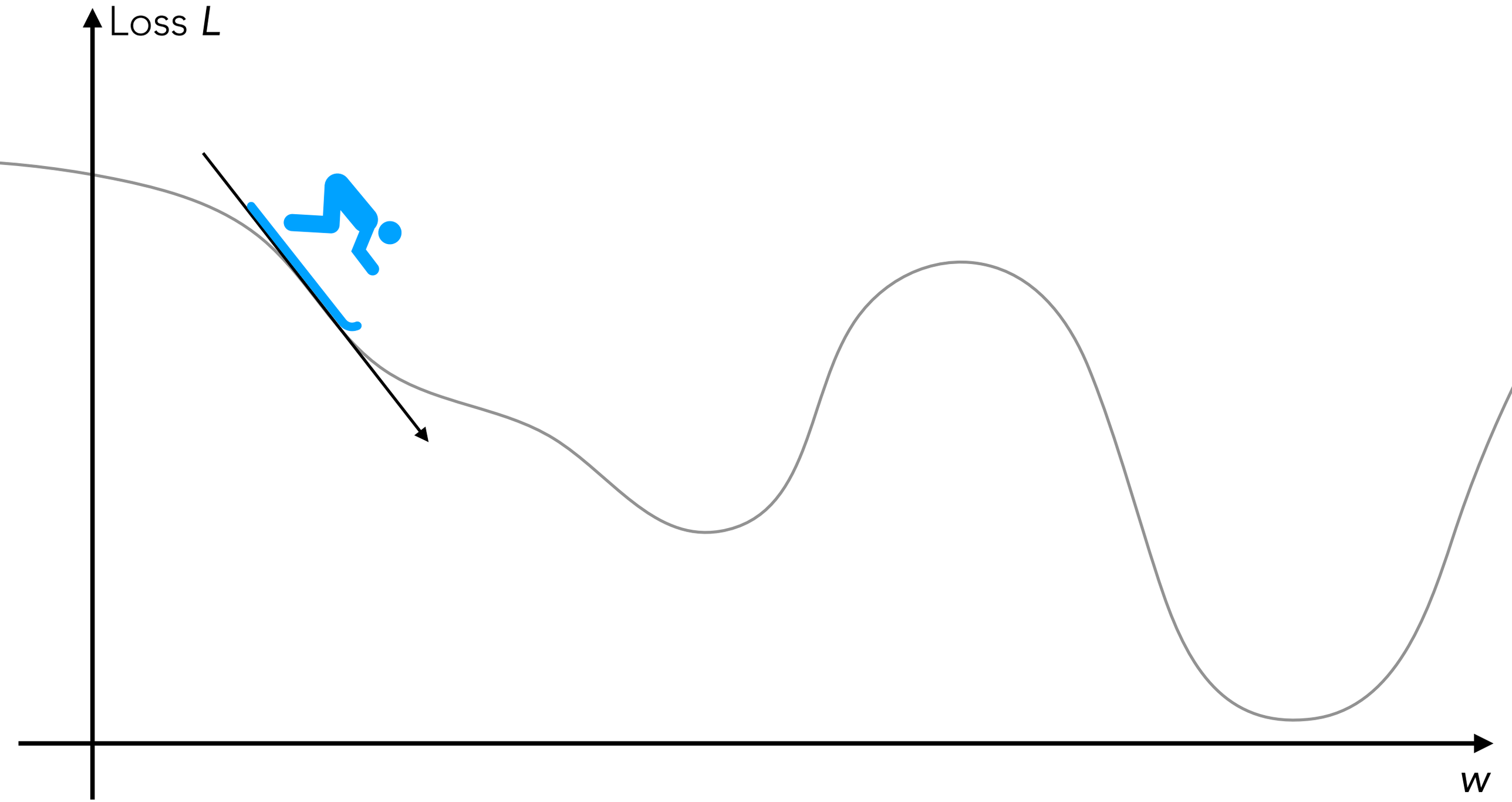
Training a Deep Net

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



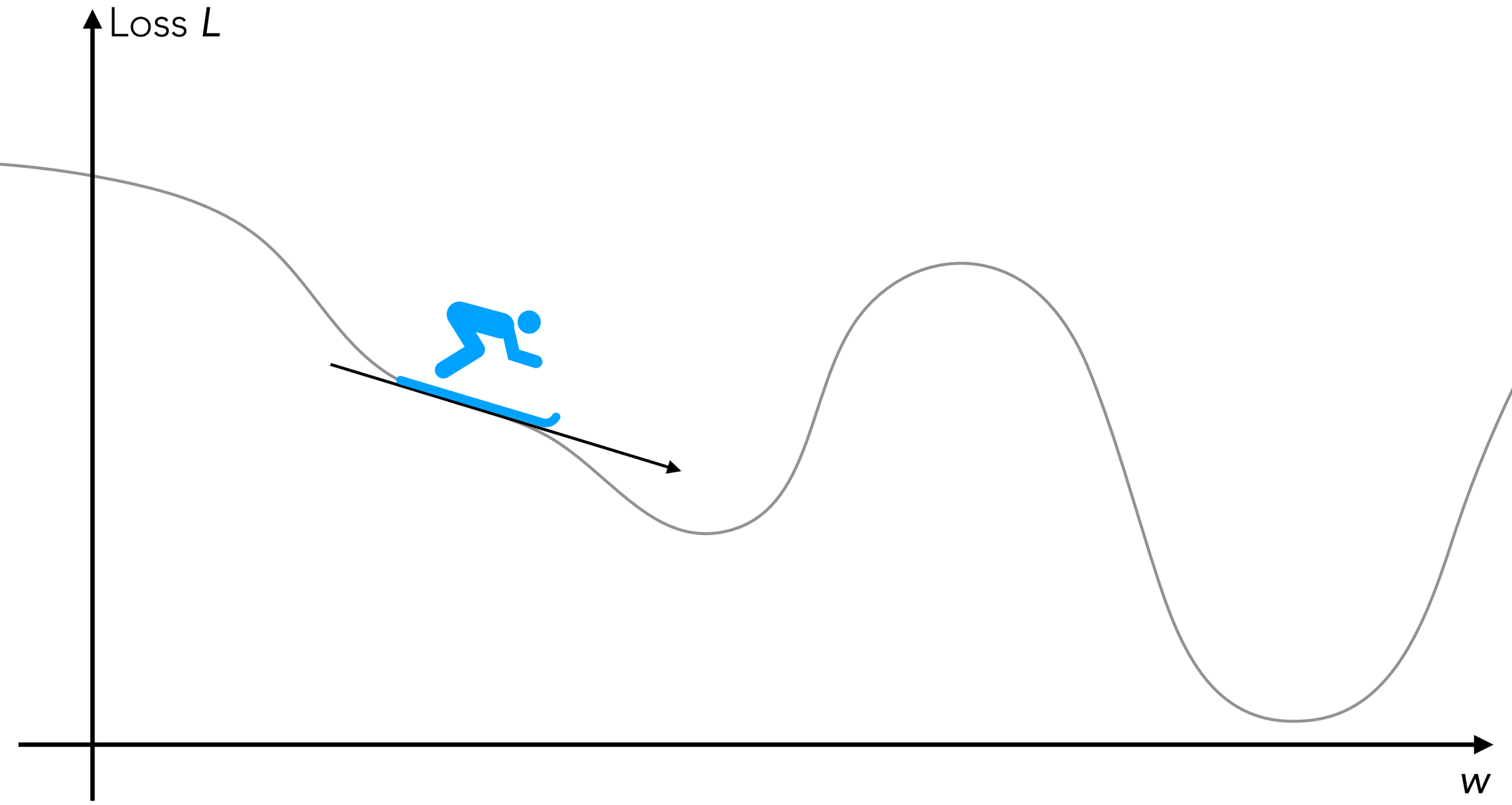
Training a Deep Net

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



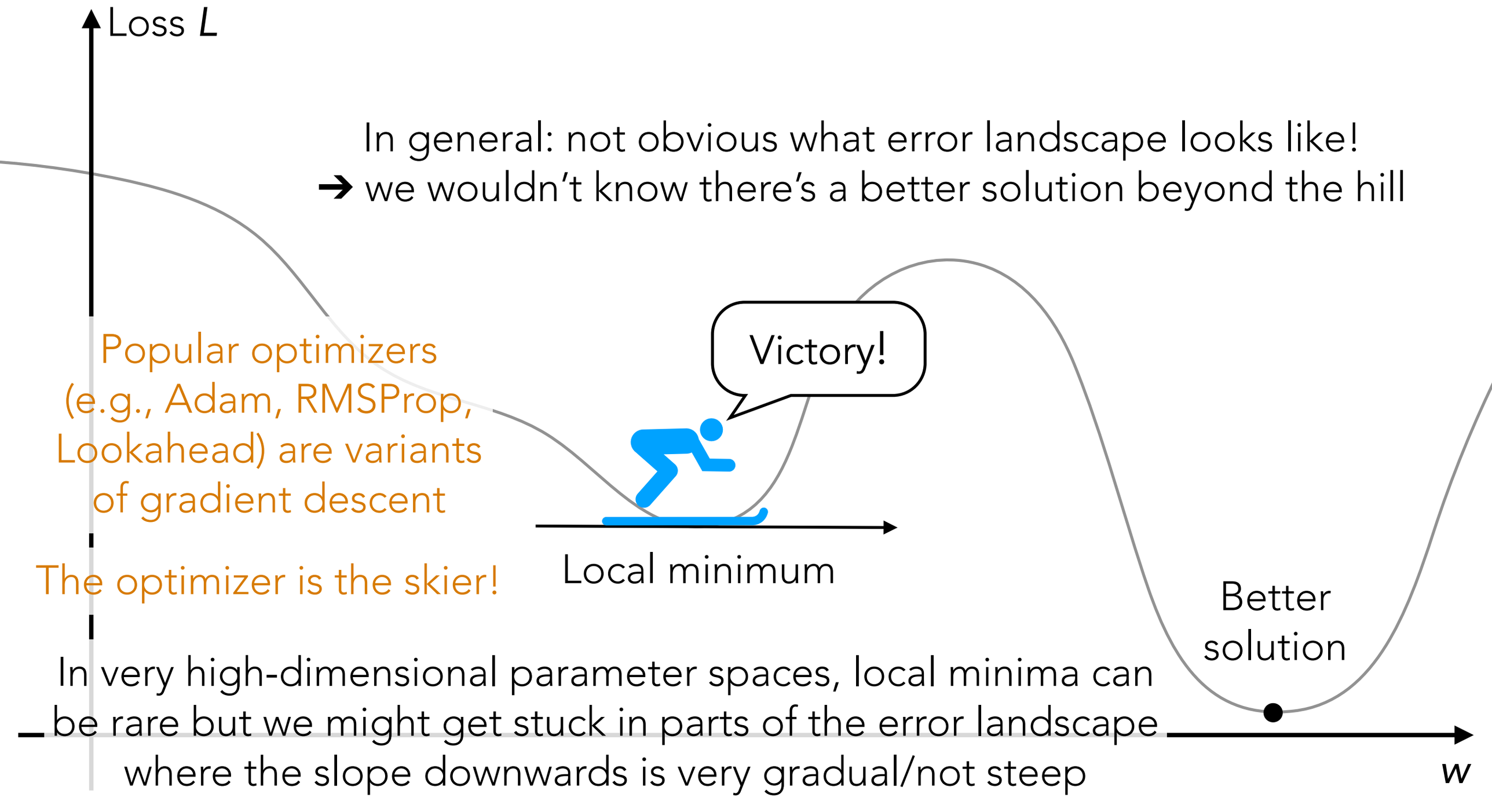
Training a Deep Net

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



Training a Deep Net

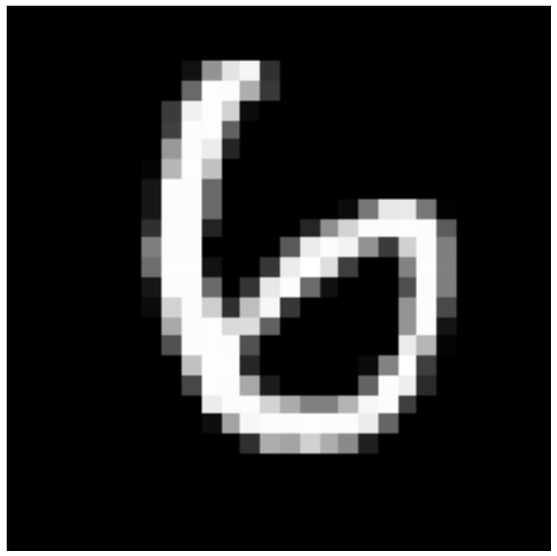
Suppose the neural network has a single real number parameter *w*



Handwritten Digit Recognition

Training label: 6

y_i



28x28 image

x_i

f_1

f_2

$f_1(x_i)$

$f_2(f_1(x_i))$

Loss

L

error

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

A neural net does function composition!

Overall loss:

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

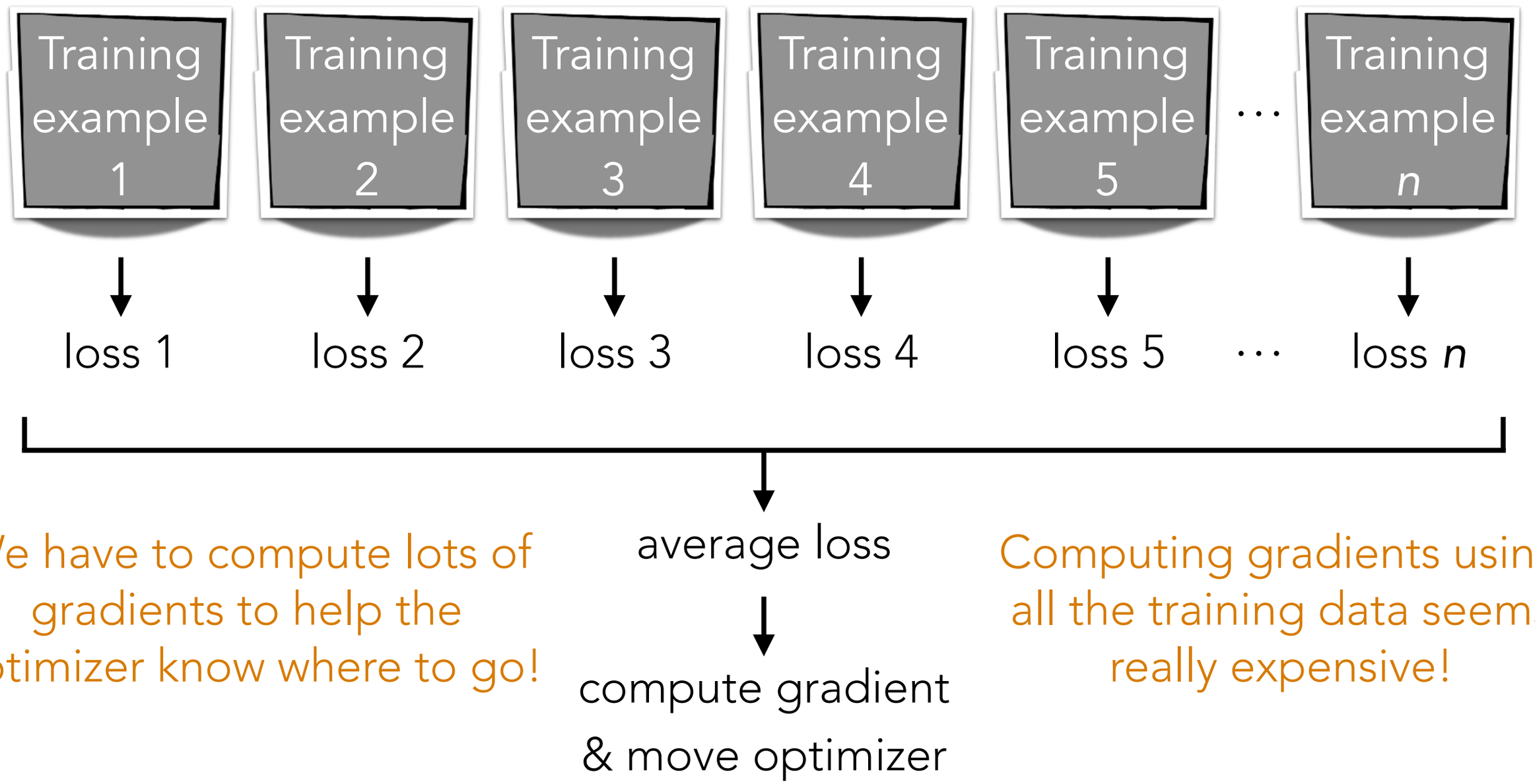
All parameters: θ

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**

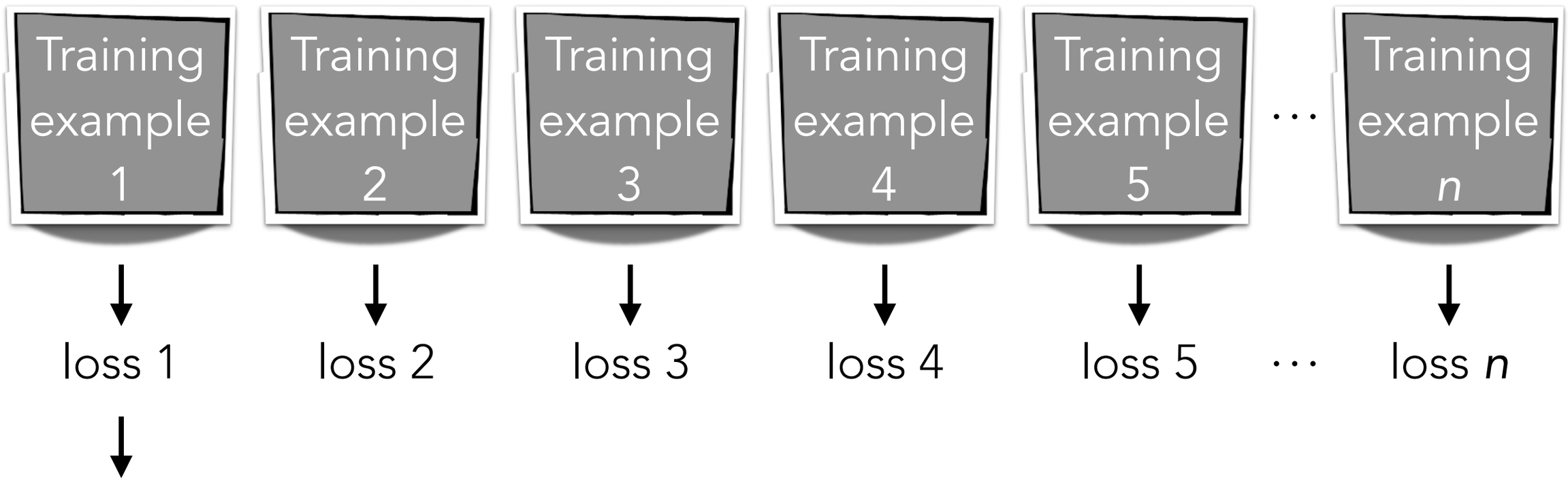
Gradient Descent



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

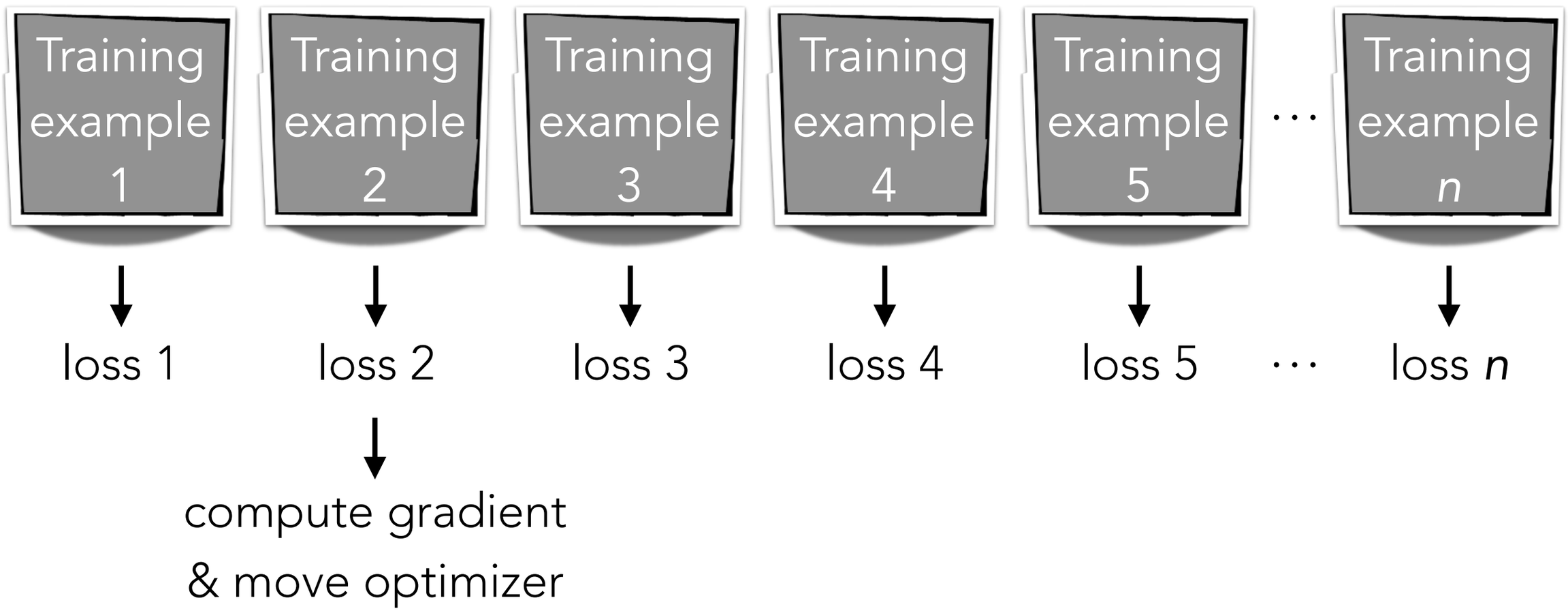
Computing gradients using all the training data seems really expensive!

Stochastic Gradient Descent (SGD)



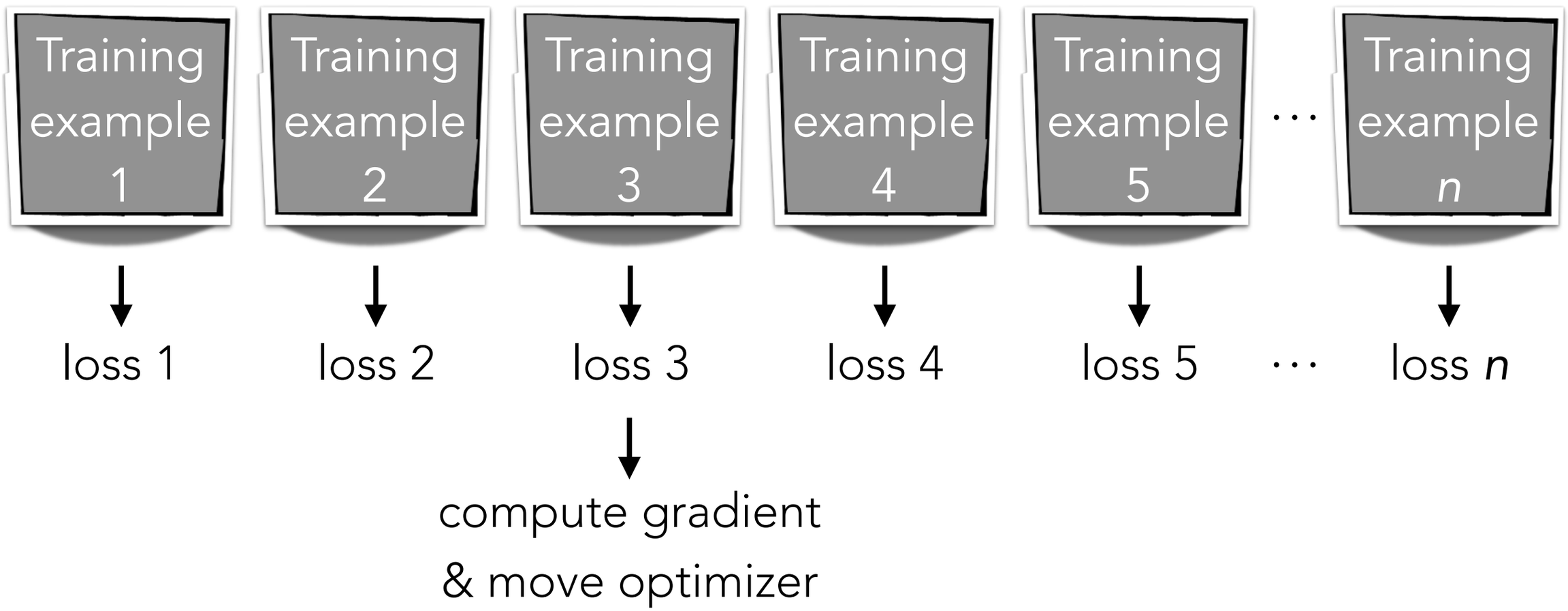
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



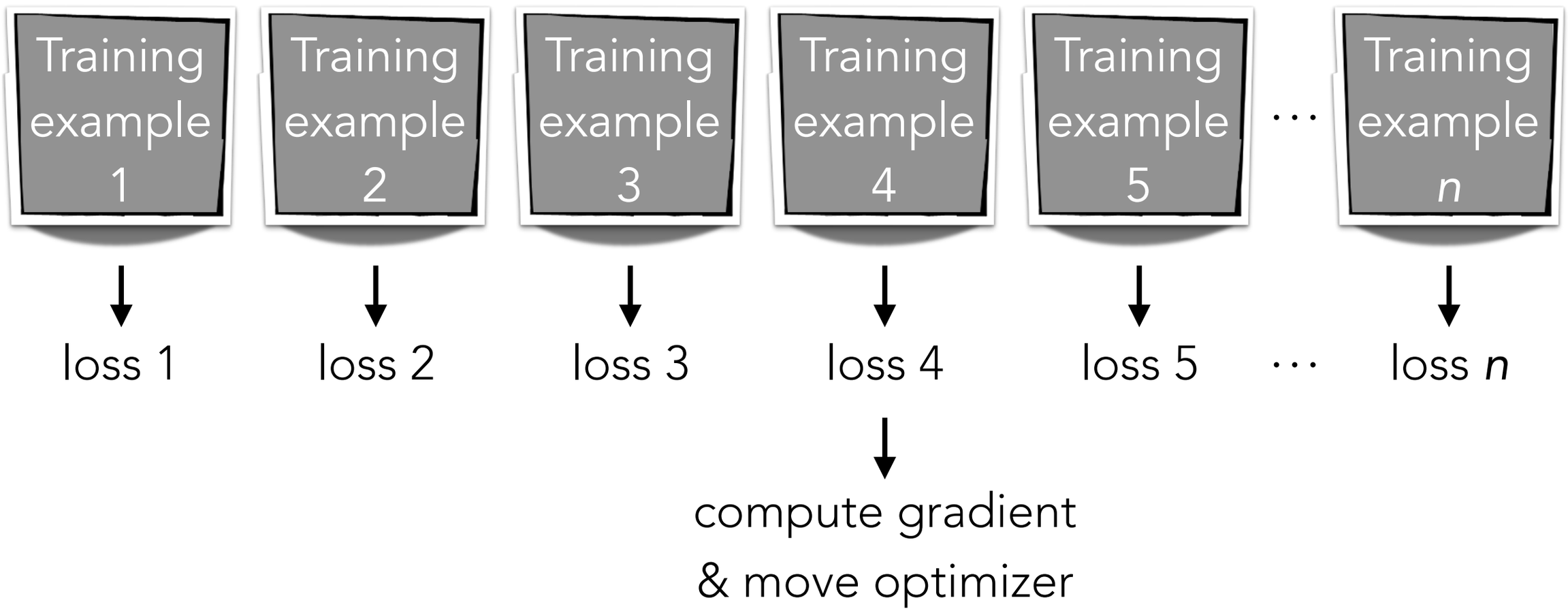
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



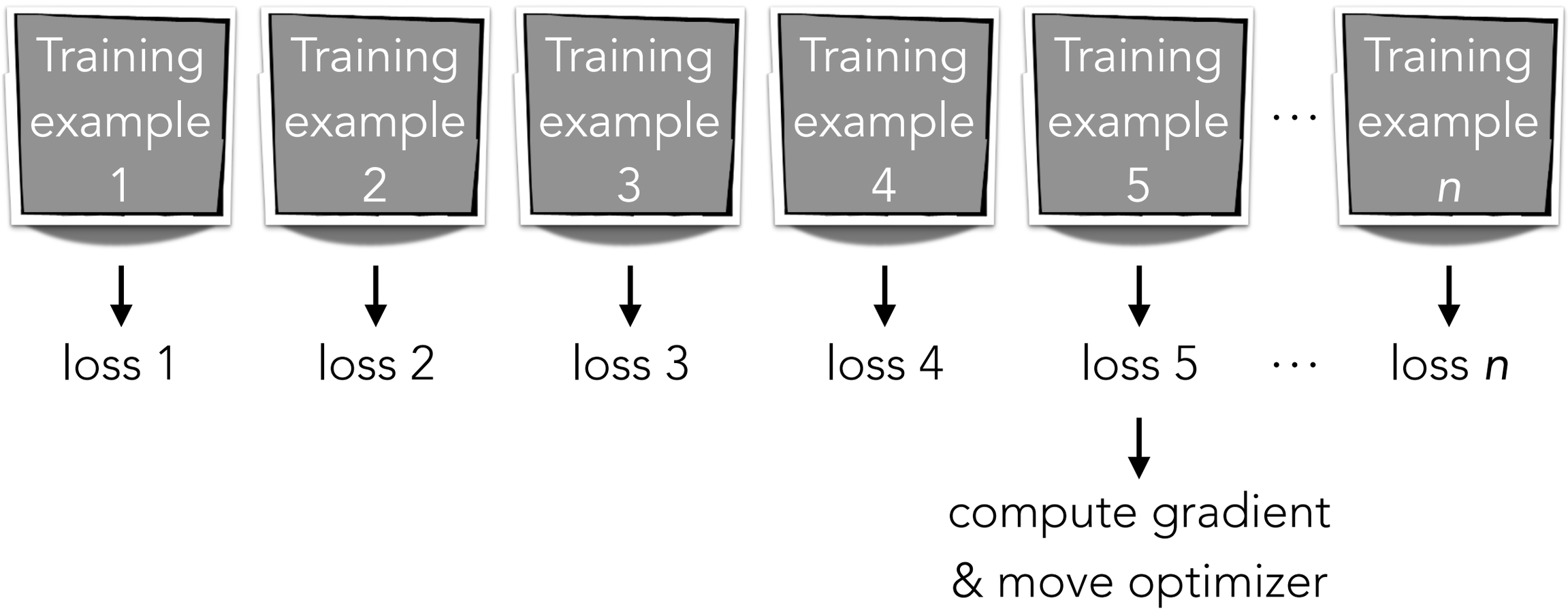
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



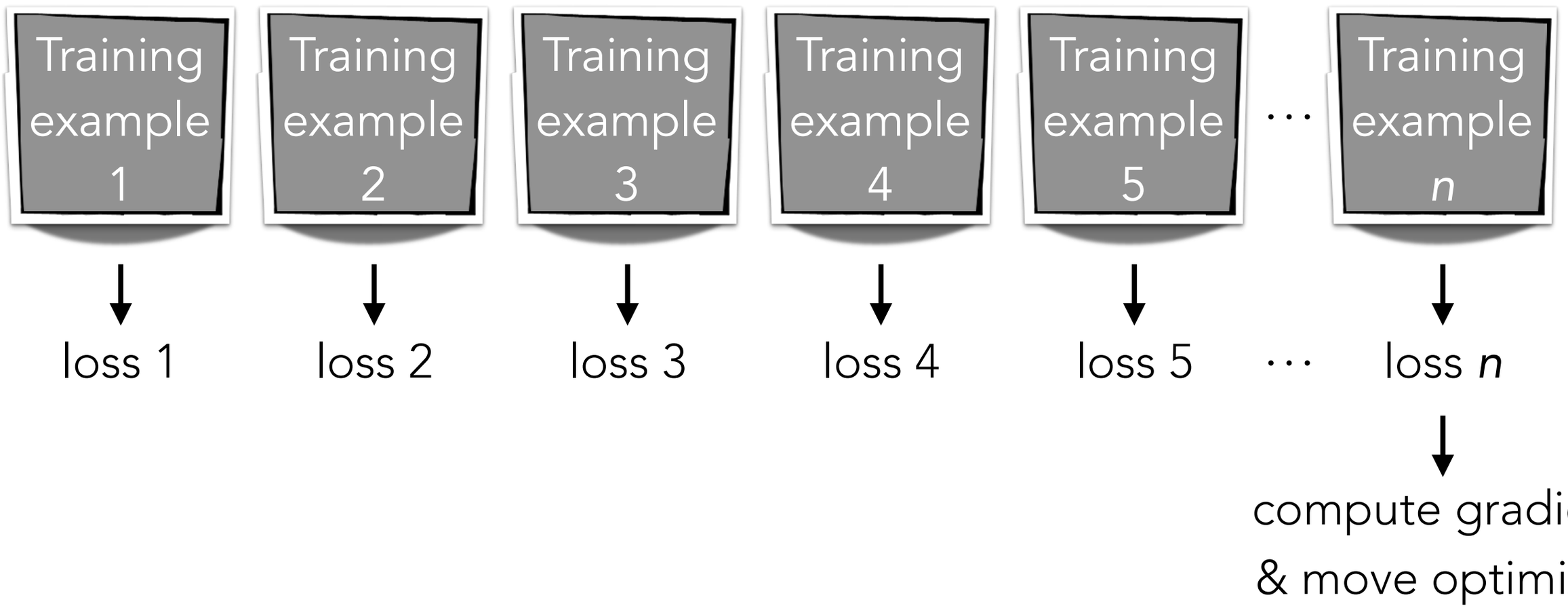
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

Stochastic Gradient Descent (SGD)



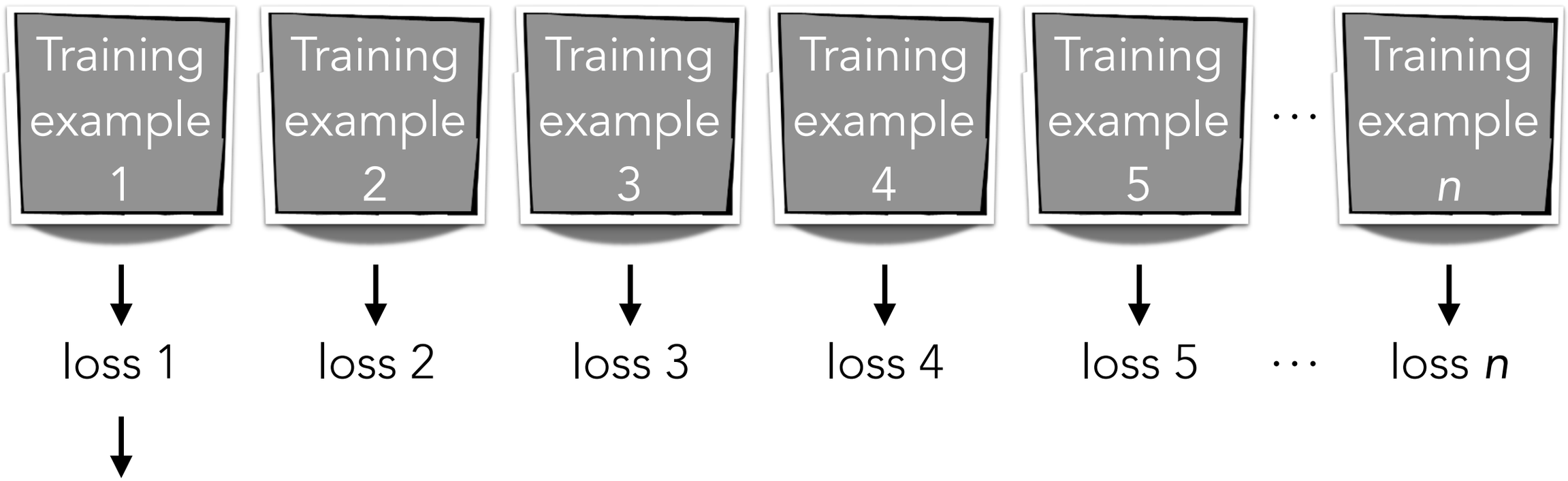
SGD: compute gradient using only 1 training example at a time
(can think of this gradient as a noisy approximation of the "full" gradient)

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)

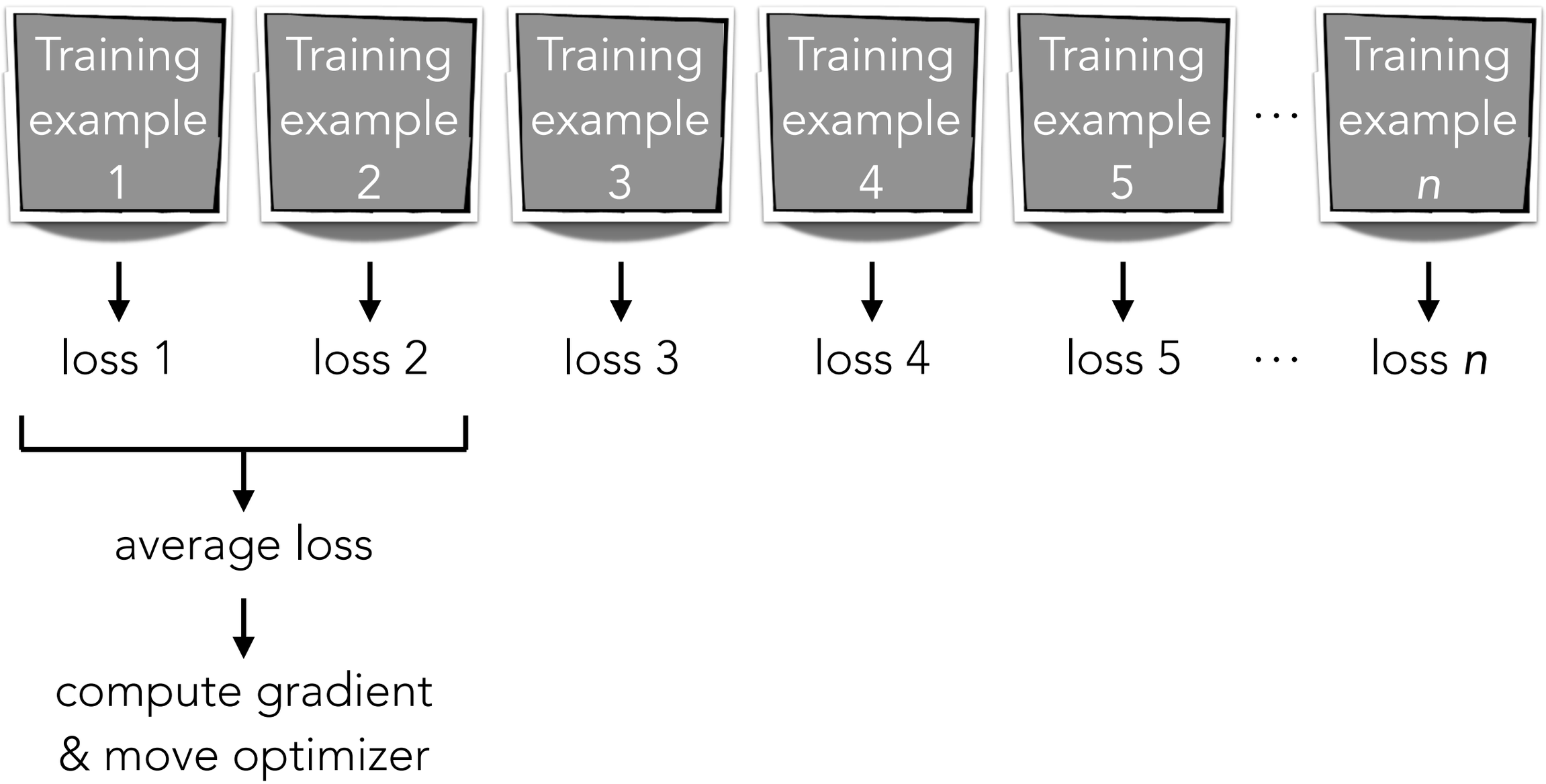
Stochastic Gradient Descent (SGD)



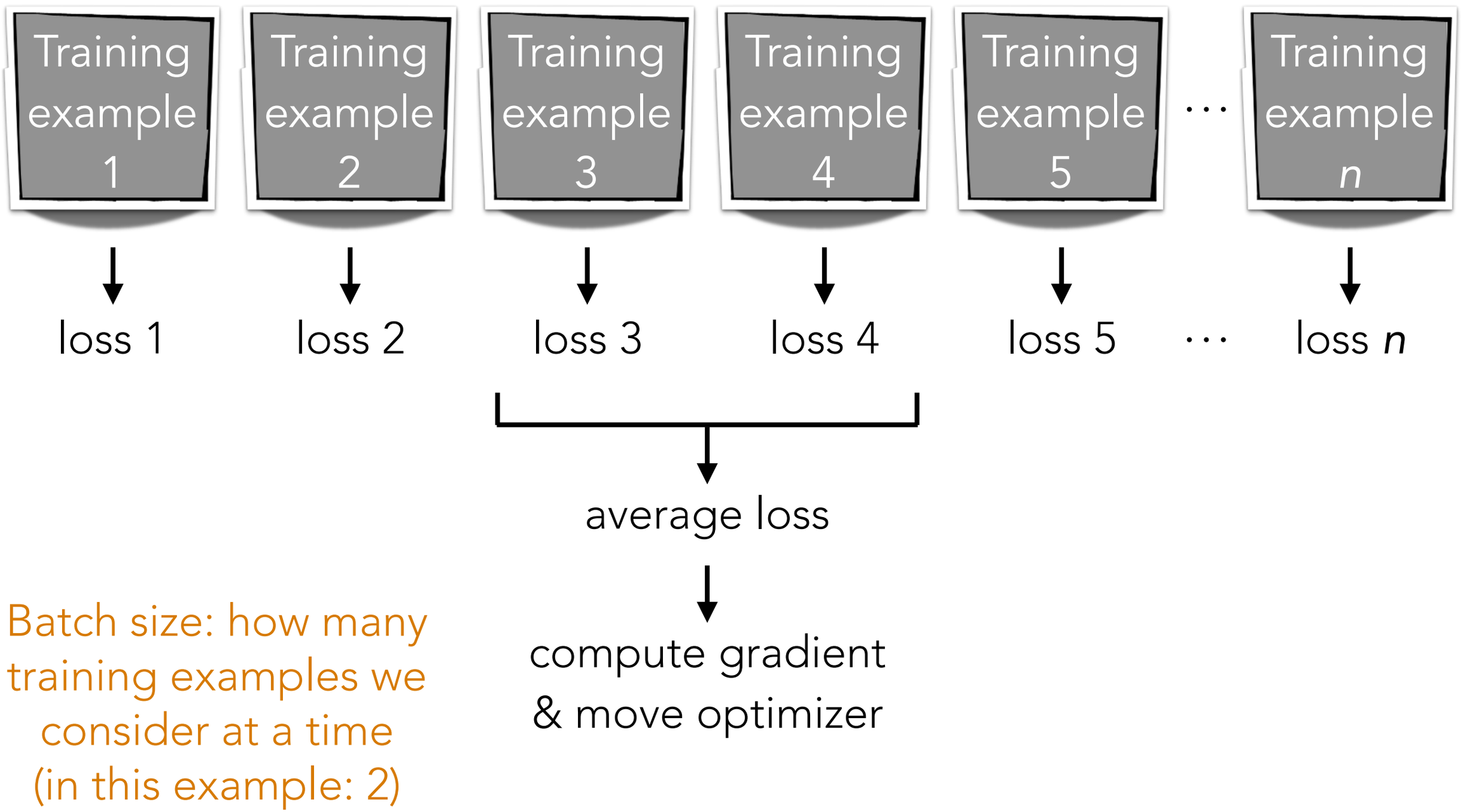
An epoch refers to 1 full pass through all the training data

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!!!