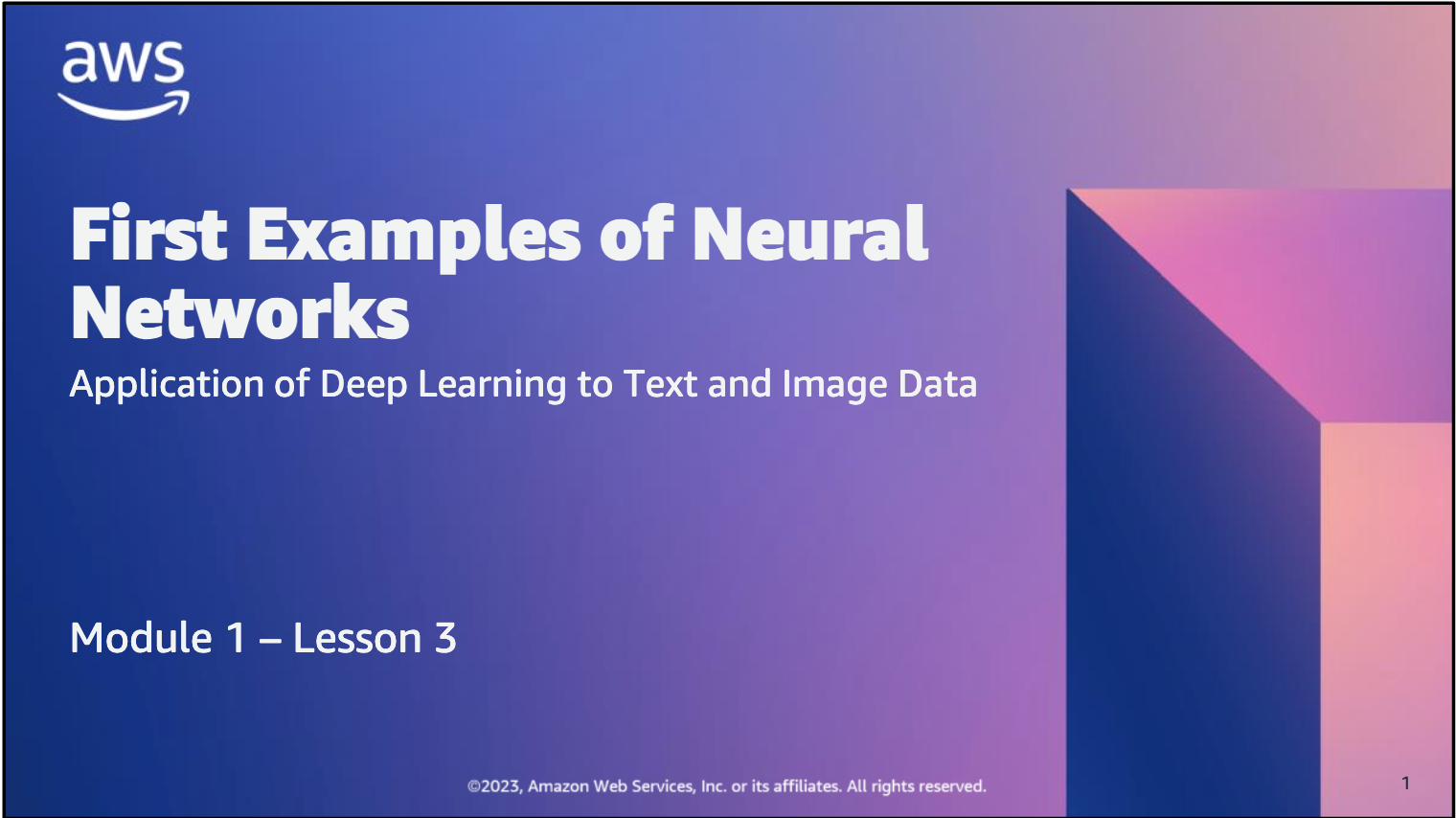
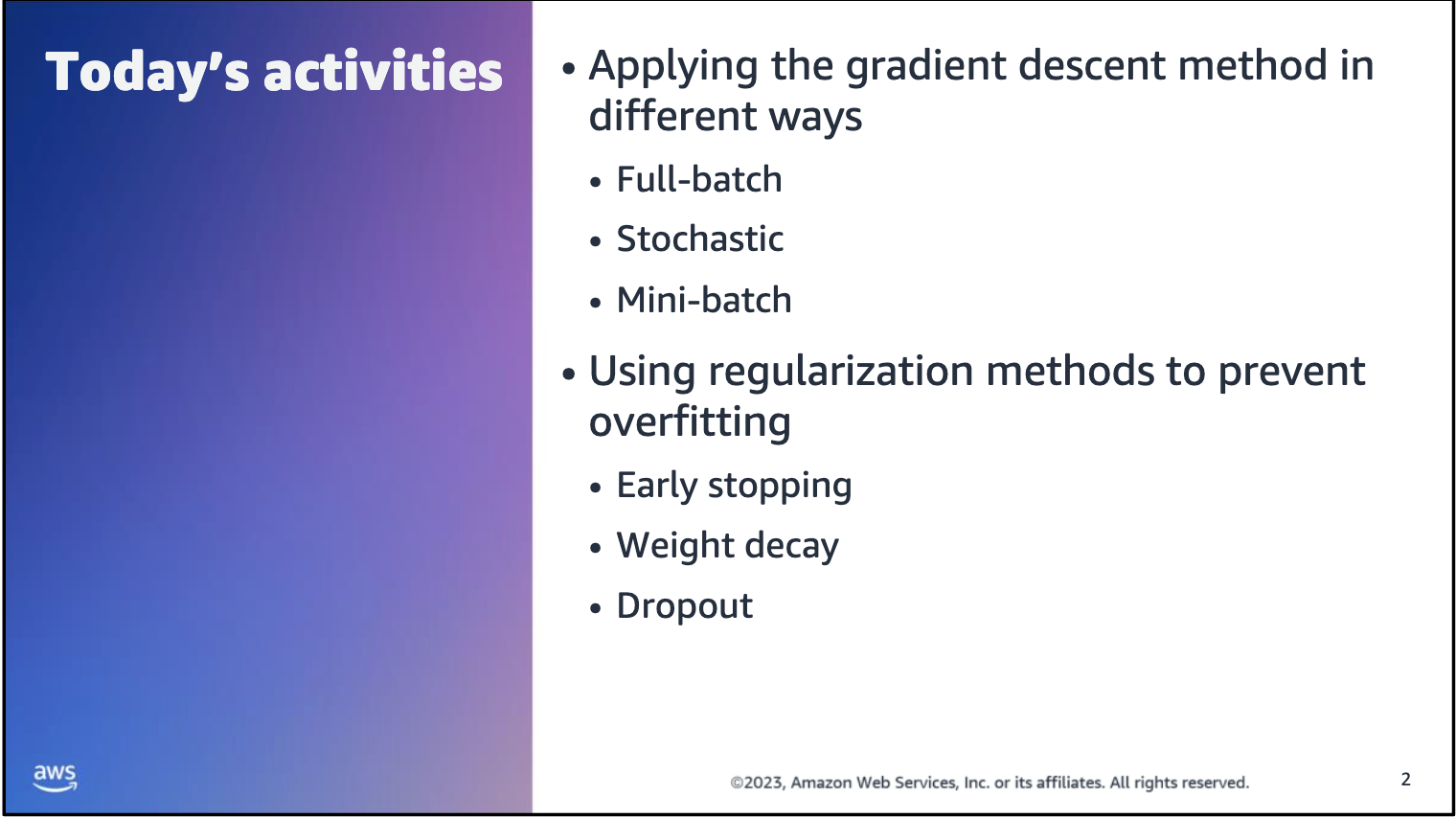
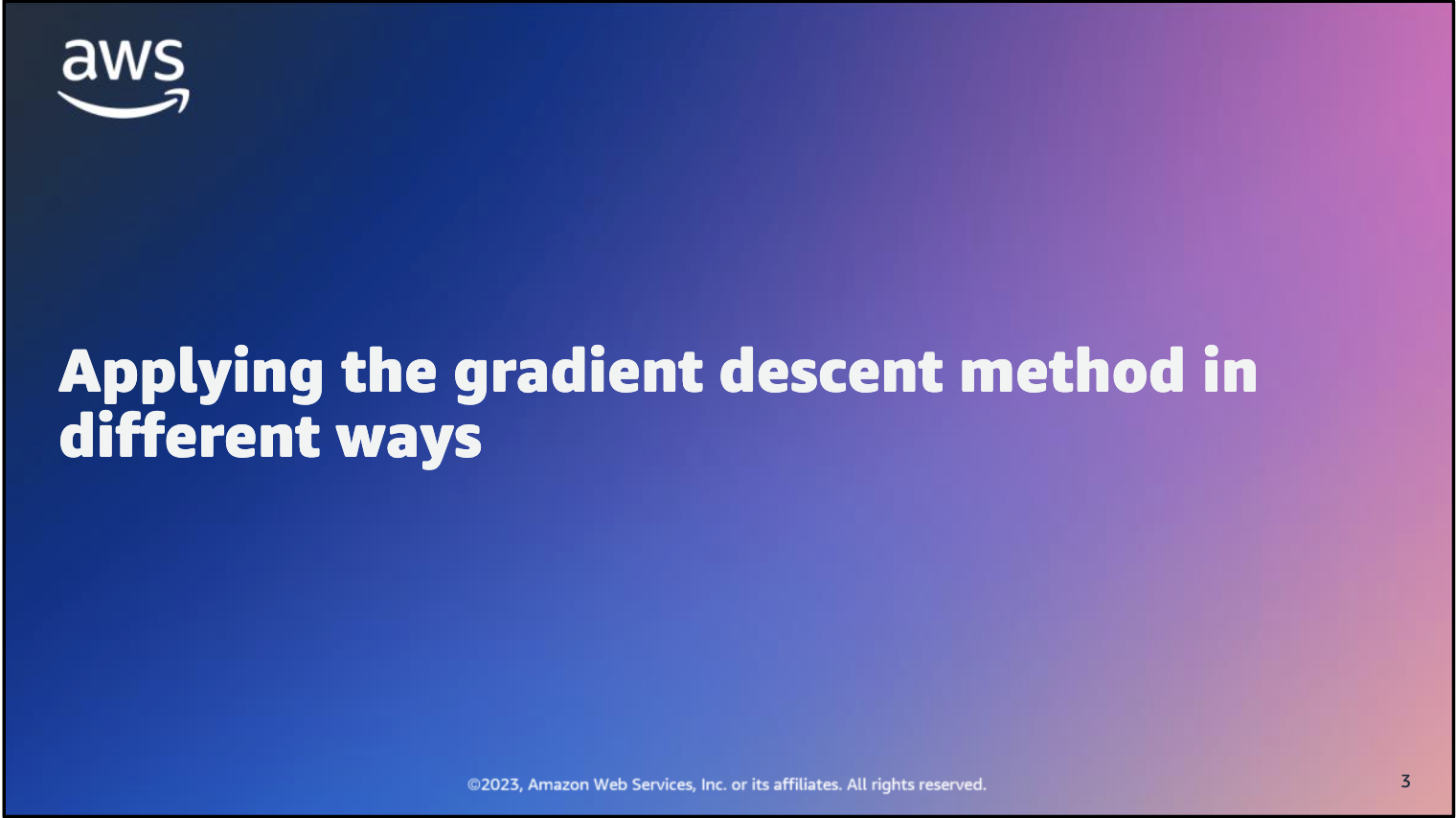
****

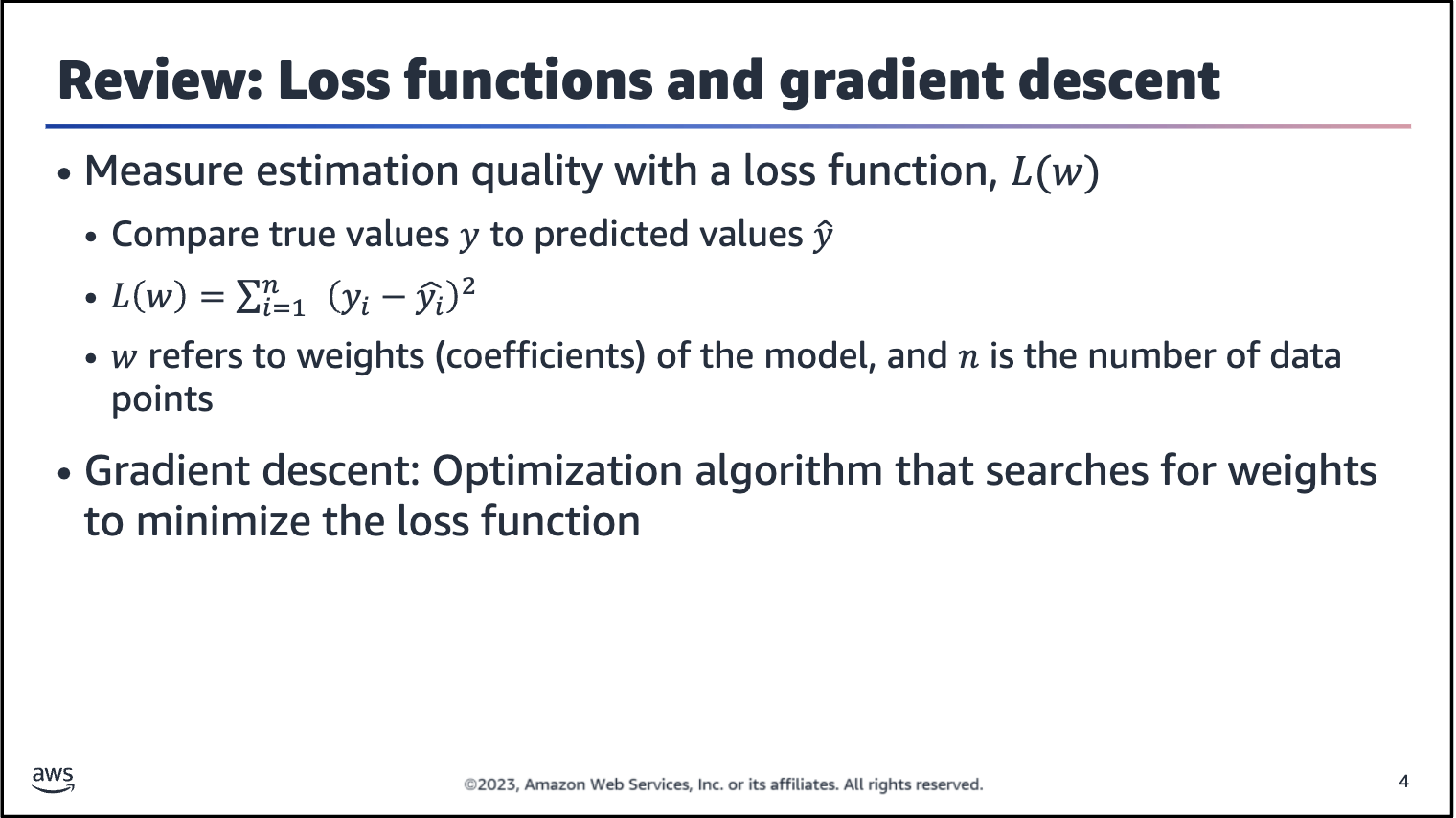
**Application of Deep Learning to Text and Image Data**

Module 1 Lesson 3 Student Guide

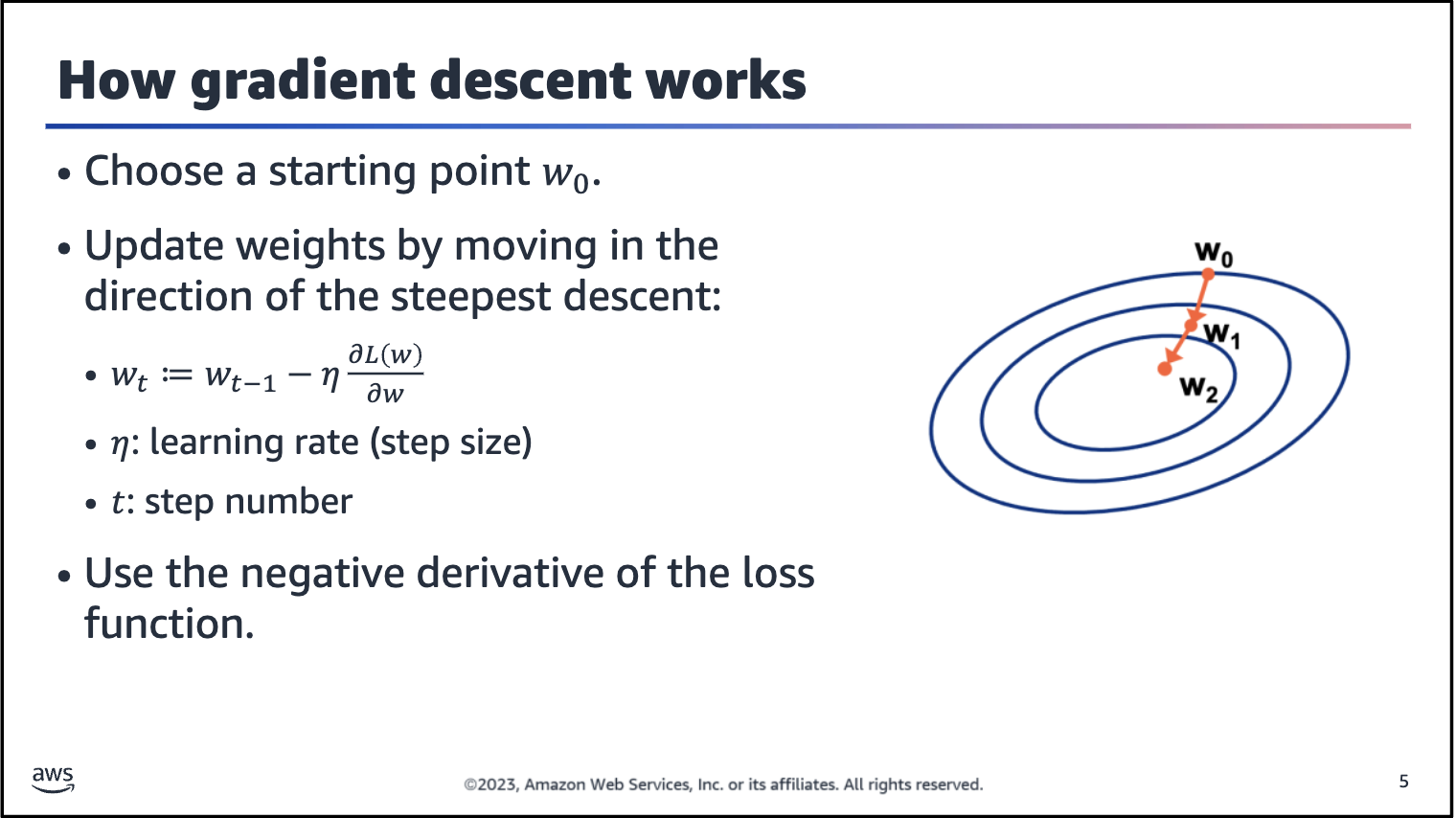






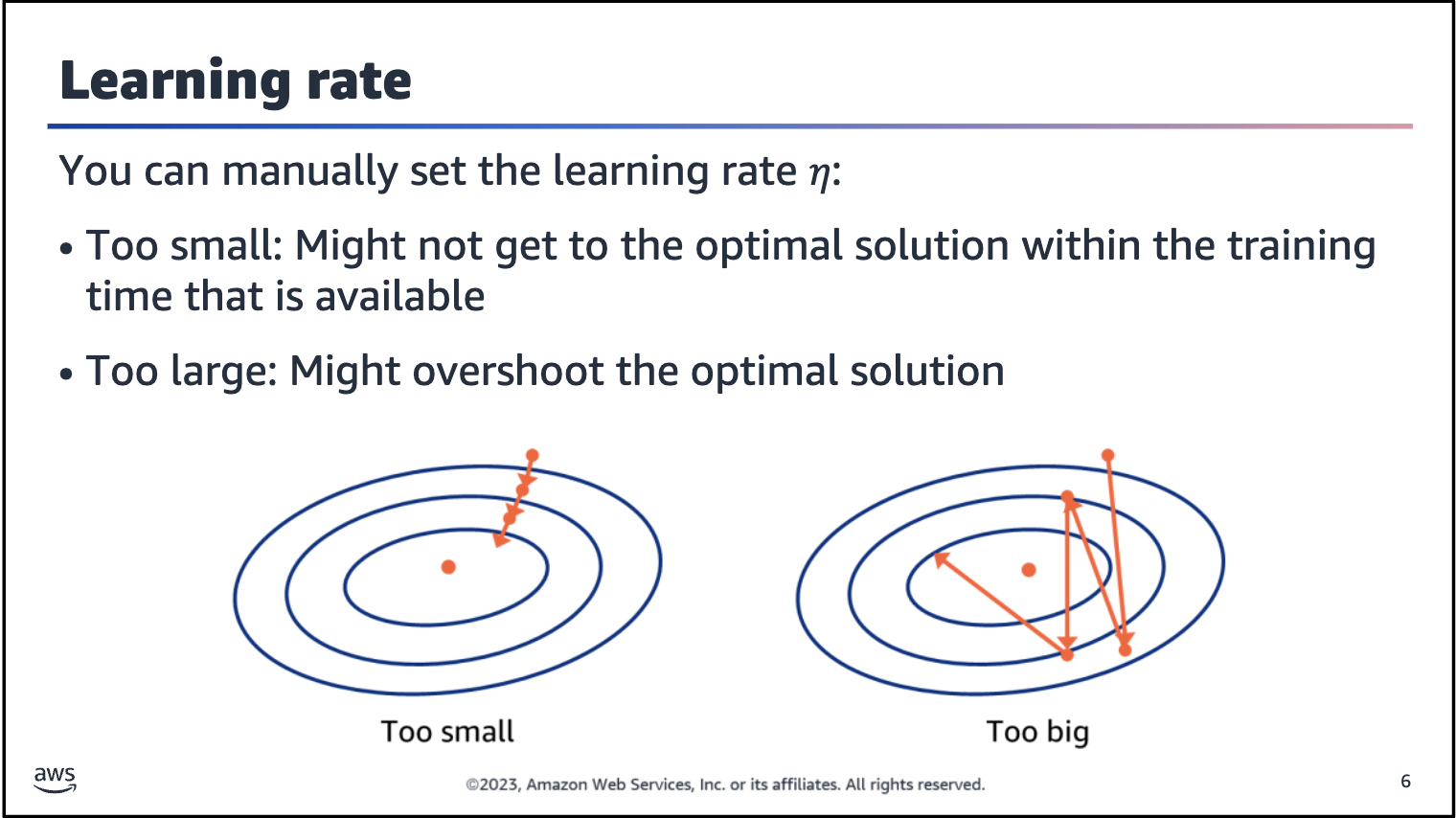


Remember: To measure the estimation quality of a network, compare the true values y to the predicted values y-hat by defining a loss function L(w).



The learning rate hyperparameter *η* specifies the step size. It starts at 1 and increases with each step. After each update, the loss gets smaller and smaller.

The negative derivative of the loss function will be used in the equation (the negative sign in the weight update equation). This allows it to move in the opposite direction of the increase.



The algorithm designer can set the learning rate *η*:

• Too small: Updates slowly, requiring more iterations to get a better solution; risk of not getting to the optimal solution

• Too large: Might overshoot the optimal solution



The objective function is usually the average of the loss functions for each example in the training dataset. For a dataset with n examples, the computing cost for each gradient descent iteration is O(*n*), which grows linearly with *n*. When the training dataset is large, the cost of gradient descent for each iteration is high.

Big O notation

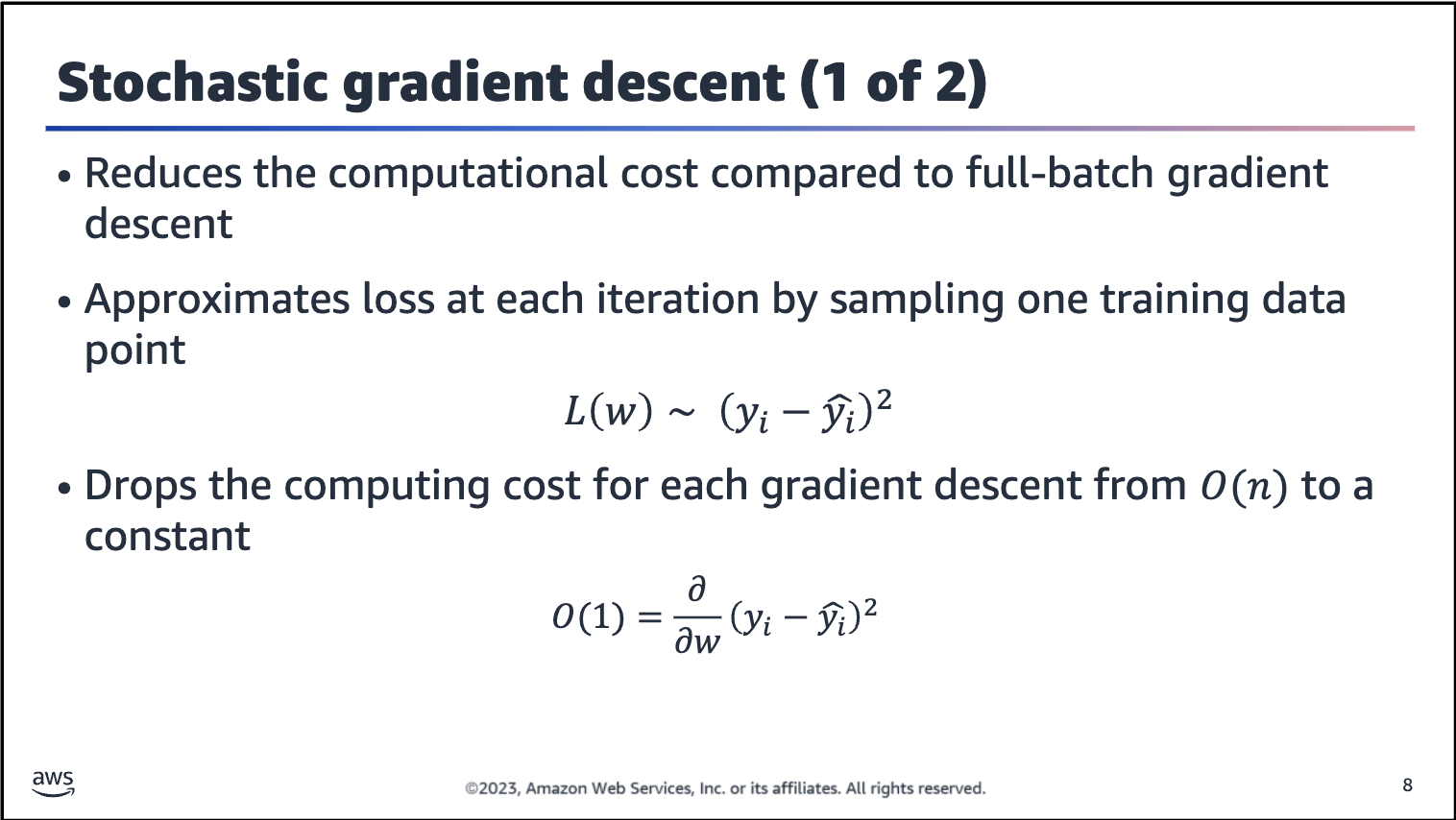
• O(*n*) is big O notation and refers to the complexity of a given algorithm. The *n* value refers to the size of the input — in this case, it's the number of items in a list.

• O(*n*)means that the algorithm will take on the order of *n* operations to insert an item (for example, looping through the list once) (or a constant number of times, such as twice or only looping through half).

• O(1) means it takes a constant time, which isn’t dependent on how many items are in the list.

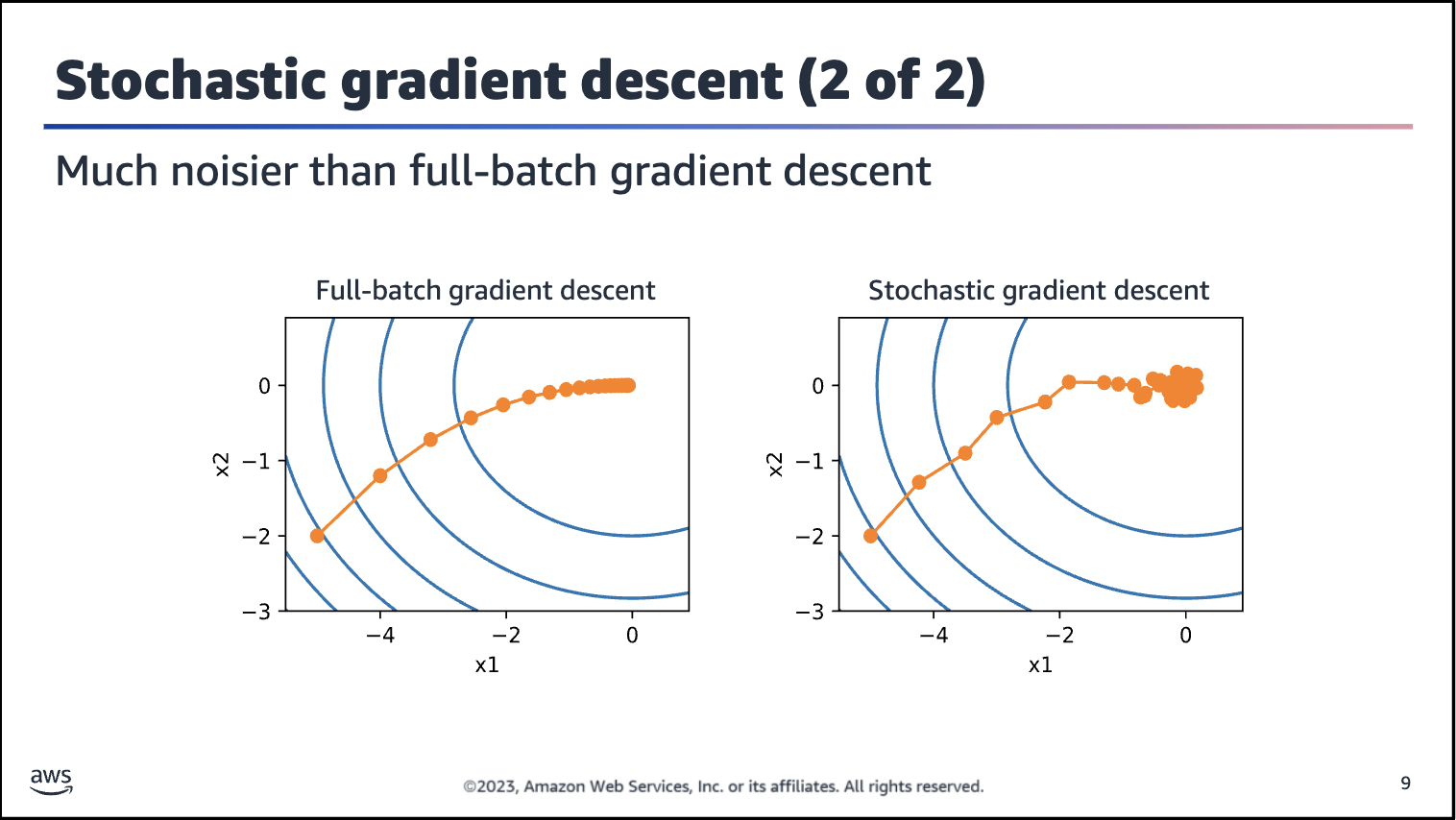
• O(n^2) means that, for every insert, it takes n\*n operations (for example, 1 operation for 1 item, 4 operations for 2 items, and 9 operations for 3 items). As you can see, O(n^2) algorithms become inefficient to handle large numbers of items.

For more information, see Big O Notation on Wikipedia



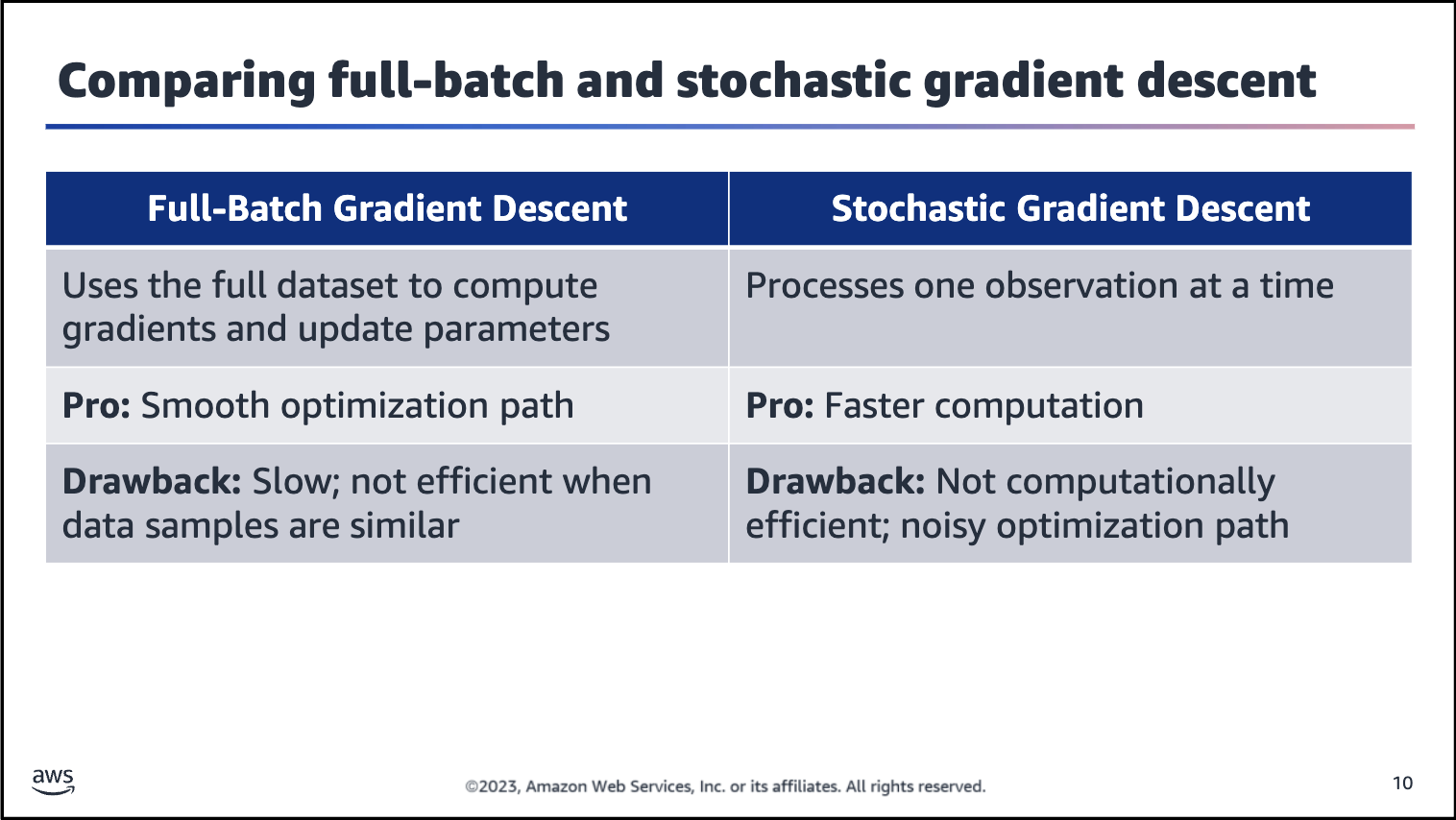
Stochastic gradient descent, which is sometimes abbreviated as SGD, reduces computational cost by randomly sampling only one training data instance to approximate the loss function at each iteration. In these formulas, the *I* represents the training data instance.

This approach improves the computational cost, but it requires you to update the weights to consider only a single data point instead of the whole training dataset (and averaging). Therefore, it’s more likely to cause a noisier weight update process (meaning more fluctuations in the loss value while training progress).

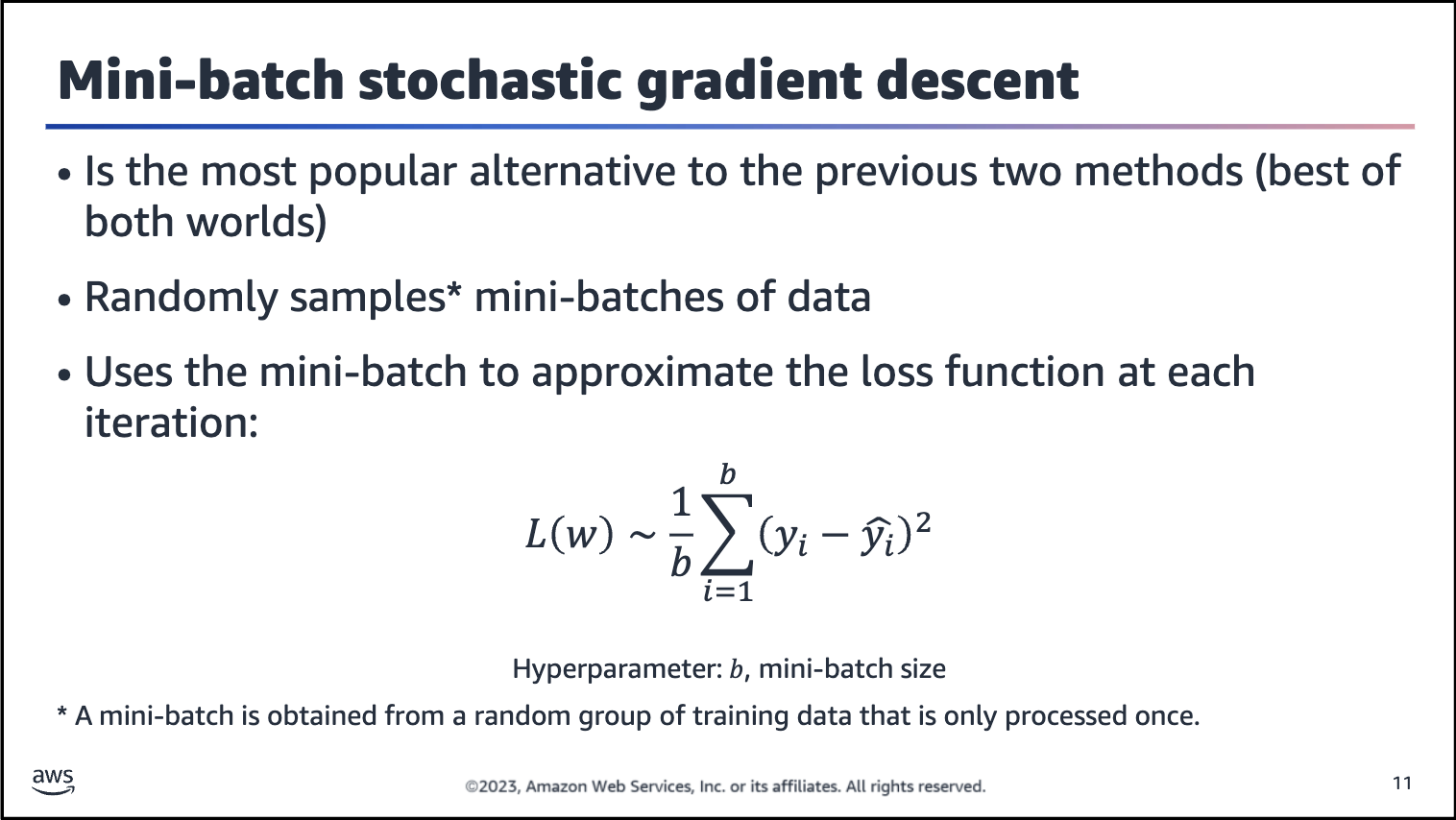


Stochastic gradient descent is likely to cause a noisier weight update process because of the lack of data points that are considered in the process. Full-batch gradient descent uses a whole dataset and averages the outcome, which removes the noise. Noise here refers to the path that is not too smooth in the stochastic gradient descent compared to the full-batch counterpart.

In the figures on the slide, x1 and x2 refer to the axes for the two weights that are being updated. Think of each dot as w1 and w2. When moved from one dot to another, both w1 and w2 change.



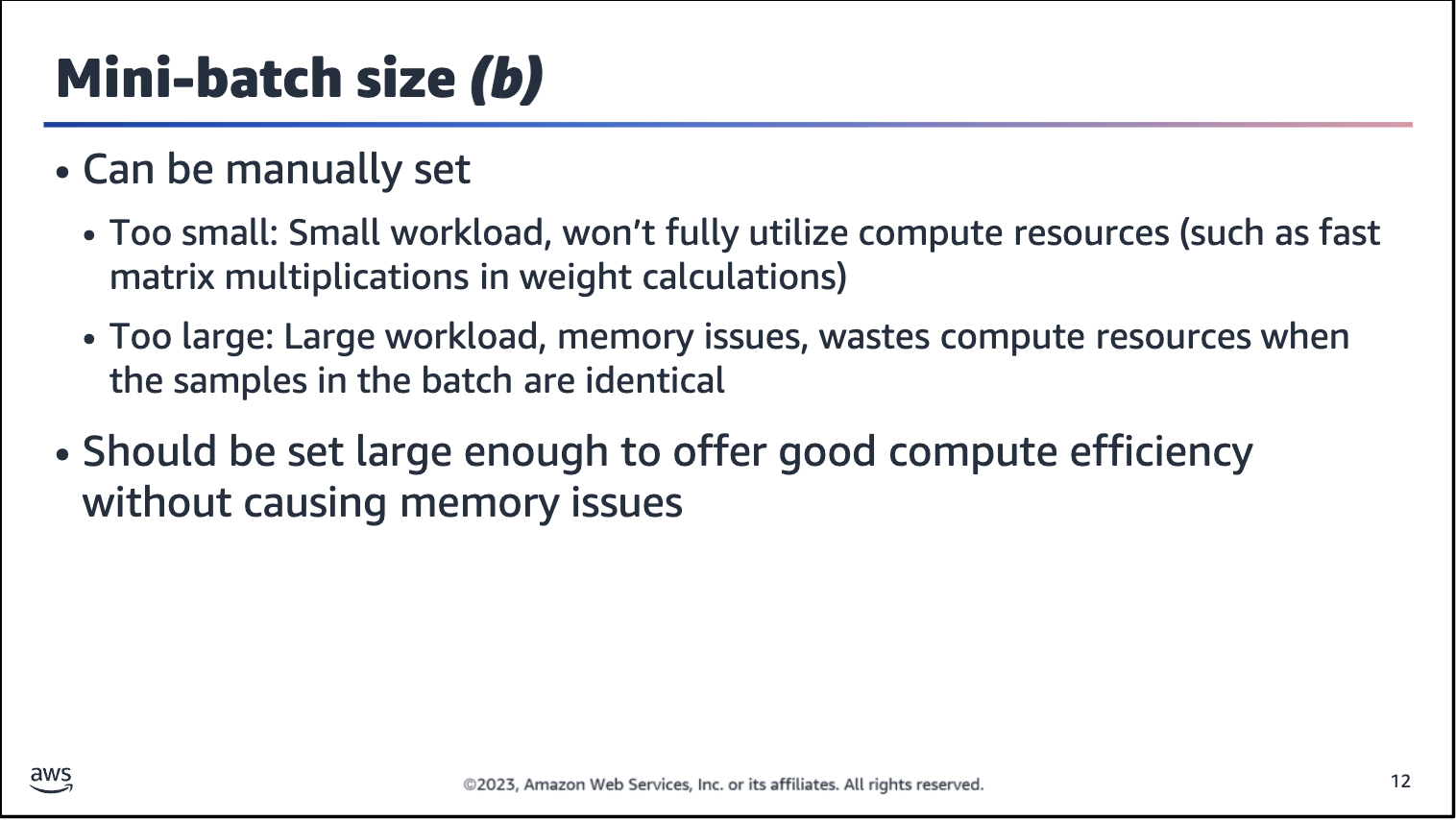
Is another option available? The next slide helps to address this.

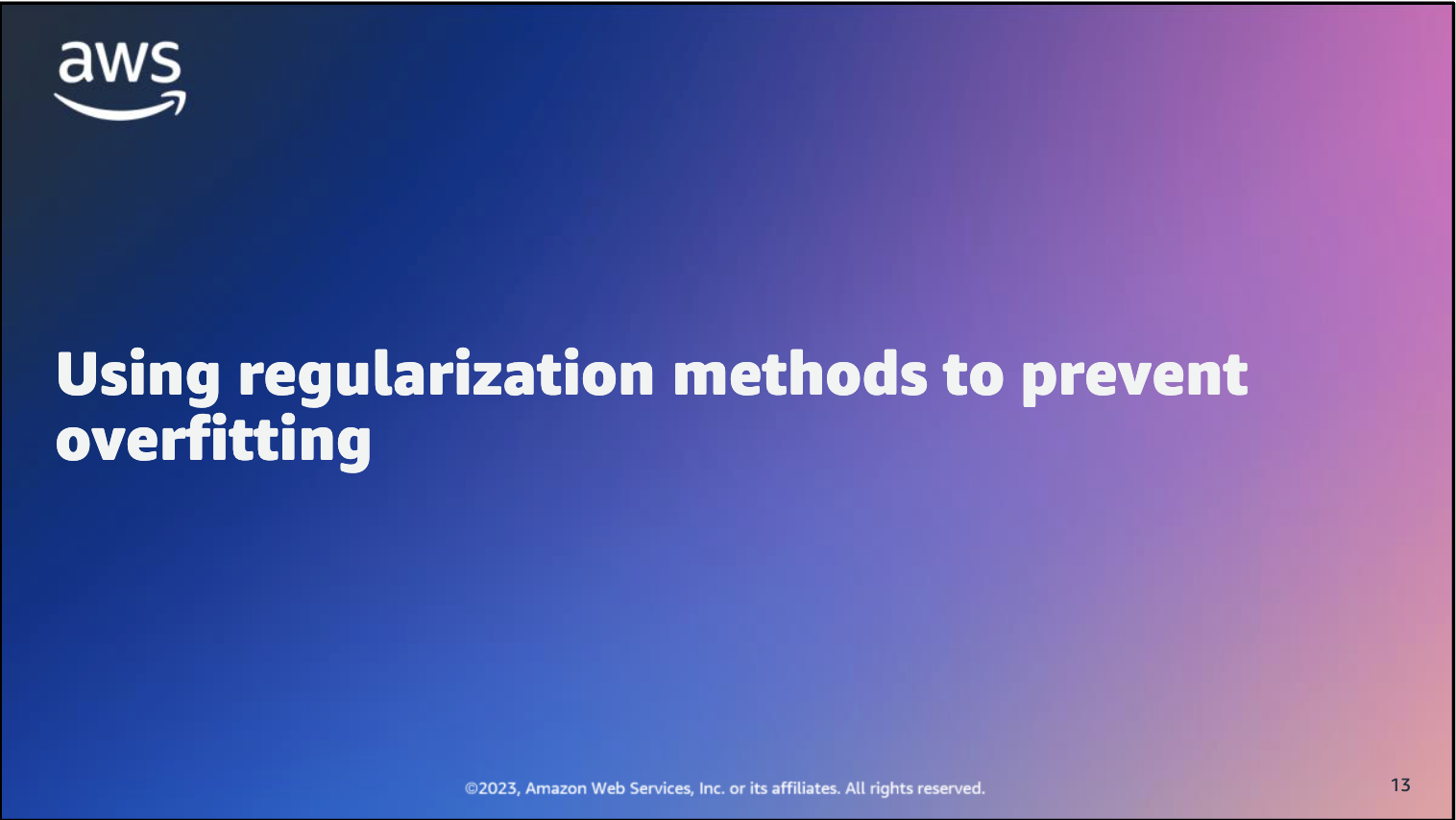


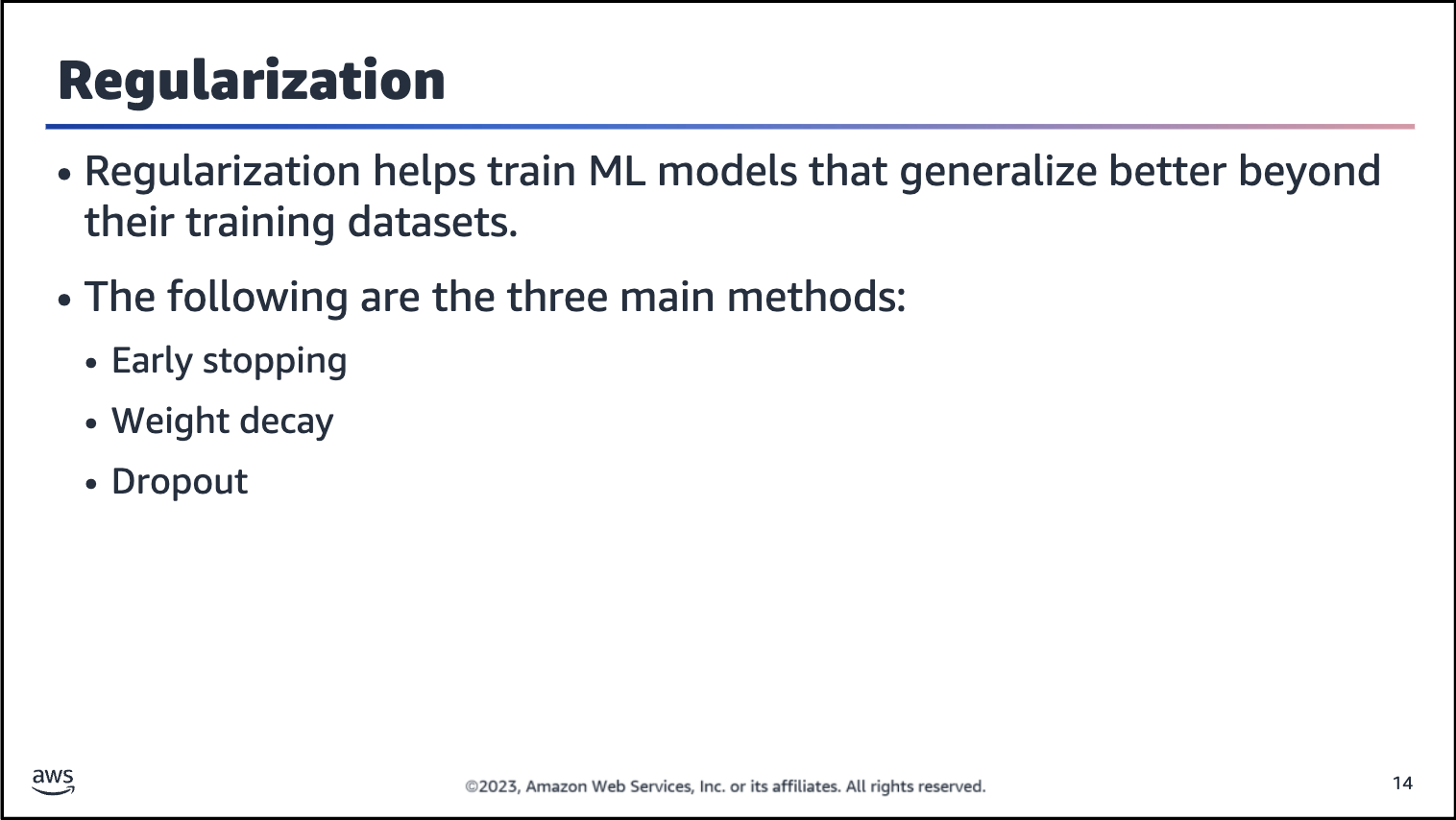
Mini-batch stochastic gradient descent increases computational efficiency by randomly sampling a mini-batch of training data instances to approximate the loss function at each iteration. A mini-batch is obtained by a random permutation of the training data — in other words, each observation is processed only once, although in random order. The gradient update over a single observation is replaced by one over the mini-batch, which stabilizes the optimization path.

The number of batches is equal to the number of iterations in one epoch. For example, if you have 2,000 training examples, you can divide the dataset of examples into batches of 500. Then, it would take 4 iterations to complete 1 epoch.

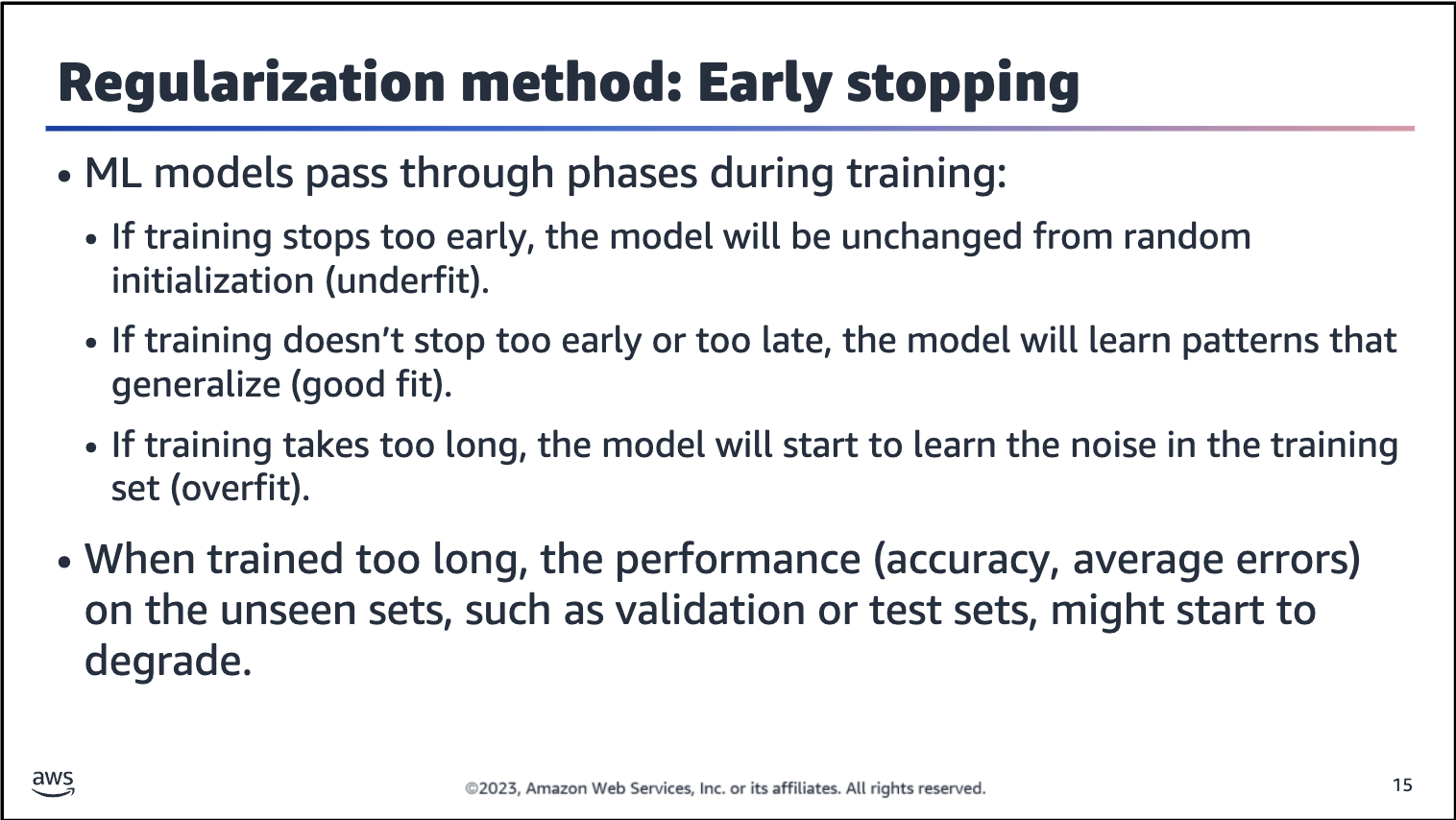
Note: Most of the time mini-batch stochastic gradient descent is referred to as mini-batch gradient descent for short.

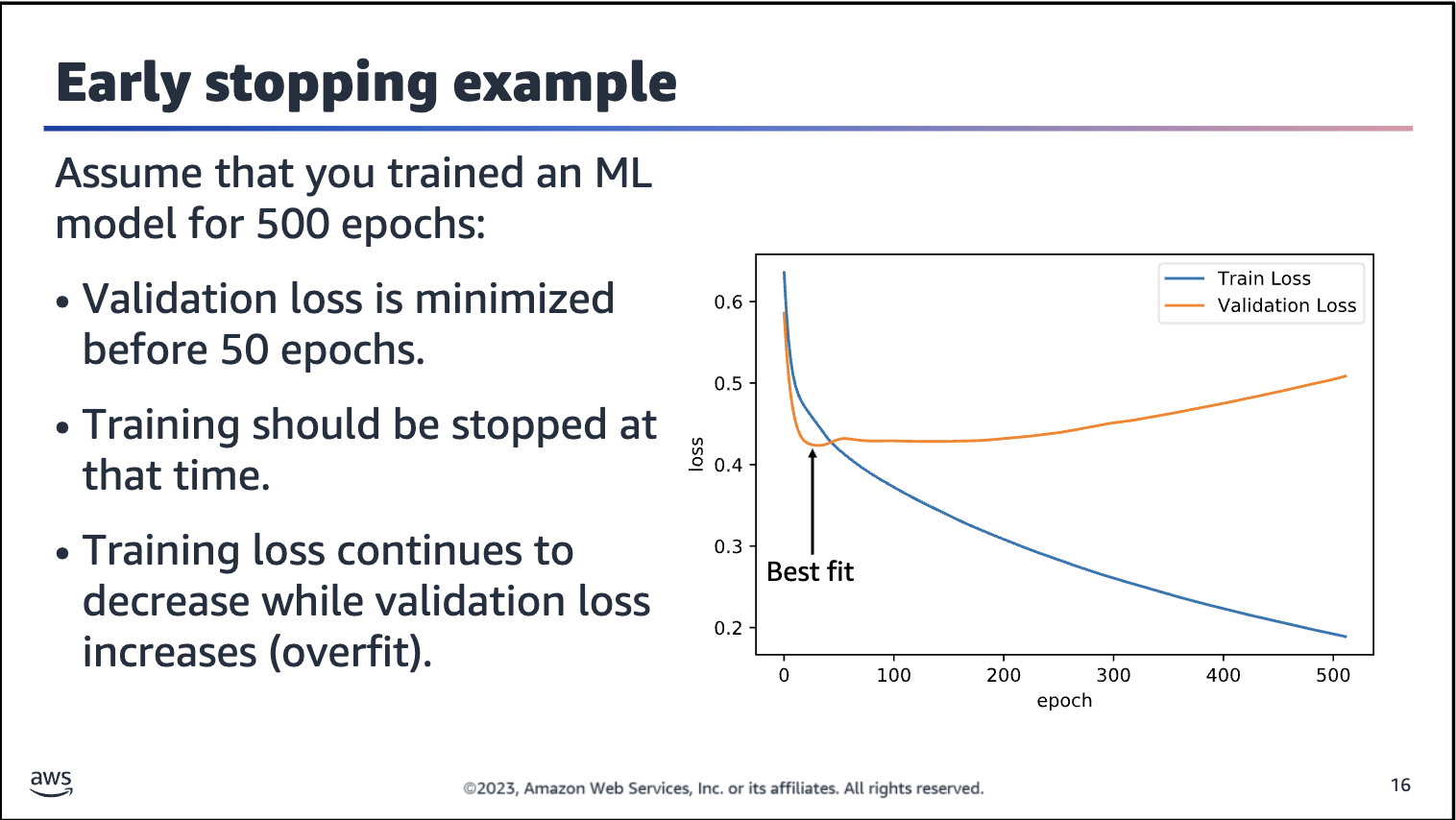






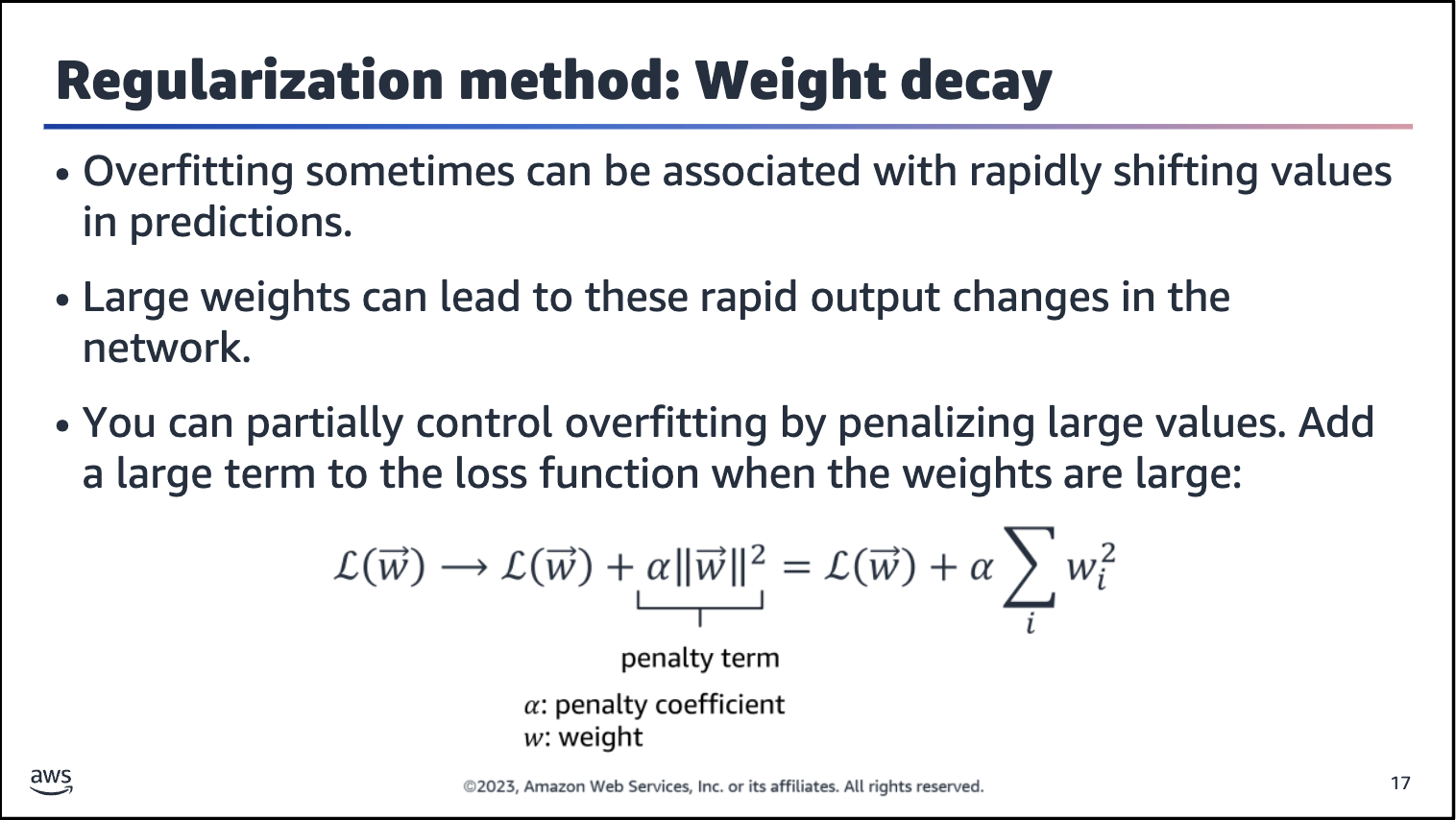
Neural networks are prone to overfitting, especially when layers are added and neurons are added to layers.





The figure on the slide shows training and validation loss as a function of epoch for a small example network. Note that the validation loss is minimized before 50 epochs, so training should be stopped at that time. Meanwhile, the training loss continues to

decrease.



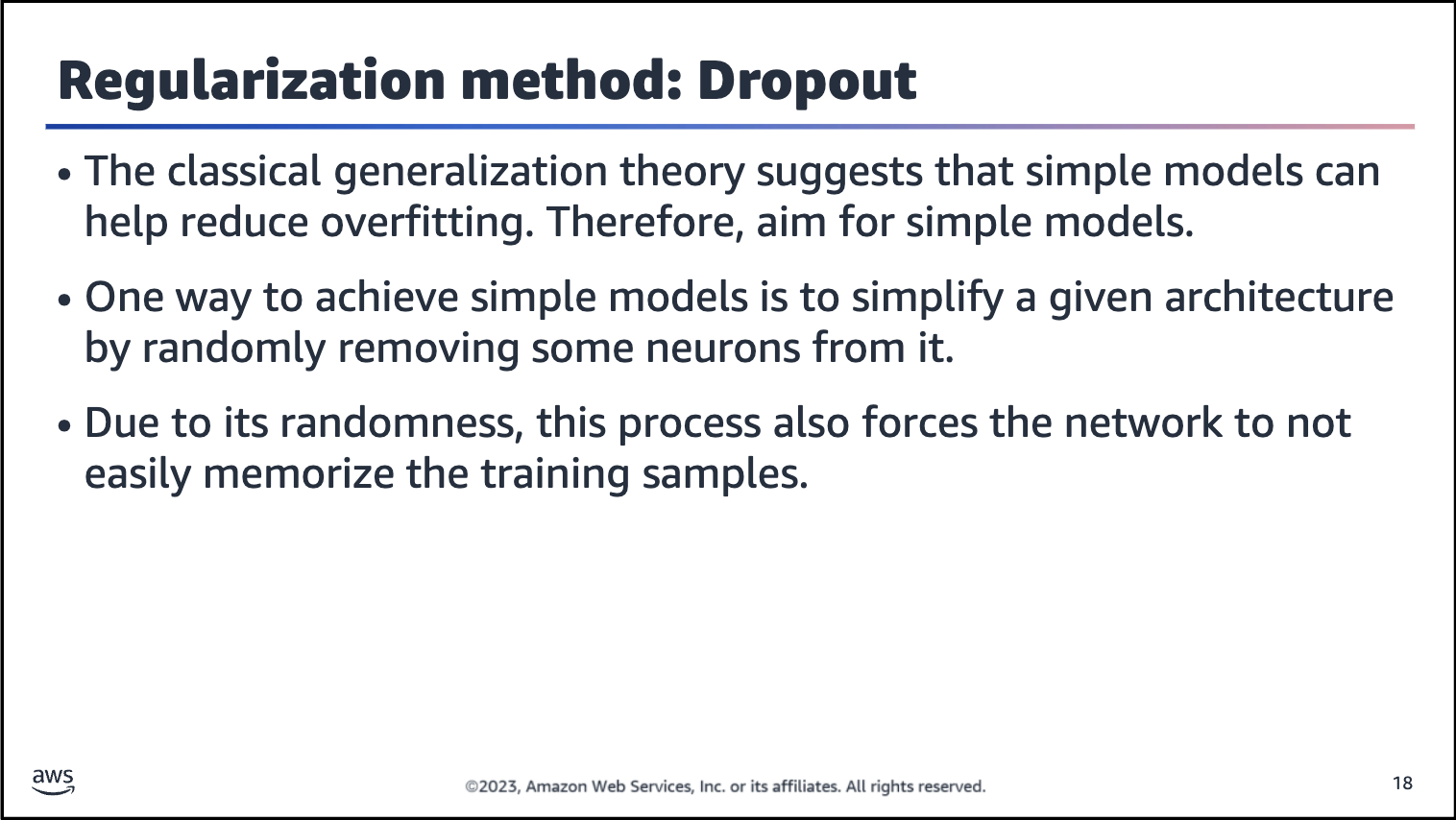
Large weights make the network unstable. Although the weight will be specialized to the training dataset, minor variation or statistical noise on the expected inputs will result in large differences in the output. Generally, you can refer to this model as

having a large variance and a small bias. In other words, the model is sensitive to the statistical noise in the training dataset.

A model with large weights is more complex than a model with smaller weights. It’s a sign of a network that might be overly specialized to training data. In practice, it’s preferred to choose the simpler models (models with smaller weights) to solve a

problem.

With weight decay, you can keep the weights under control. The added term has a penalty coefficient and summation of squared weights. When weights change quickly and in large amounts, the model can overfit the data. An added penalty term makes it more difficult to overfit.



Dropout is one of the most popular regularization methods when training neural networks. This method randomly disables or removes neurons from a network.

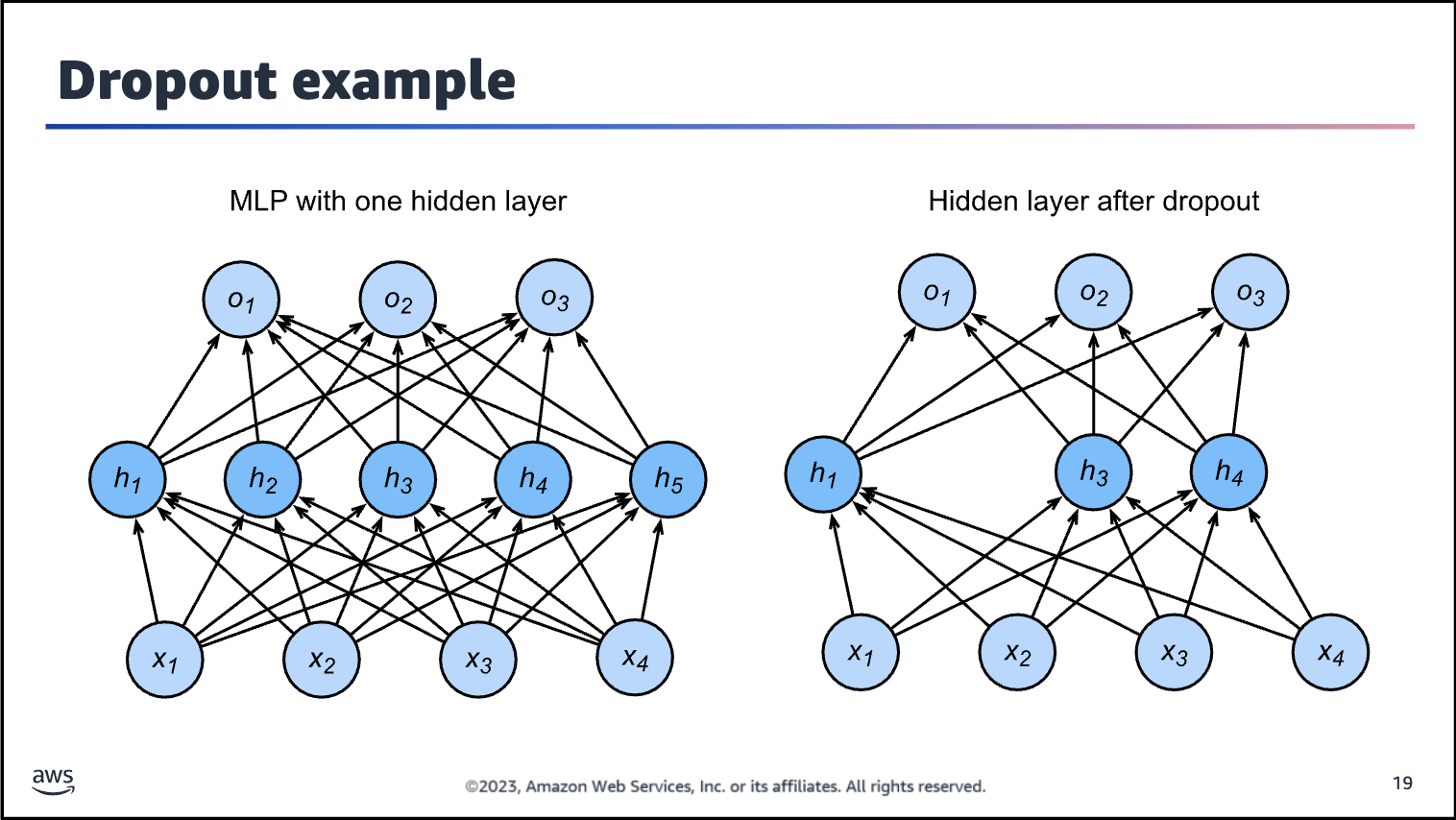
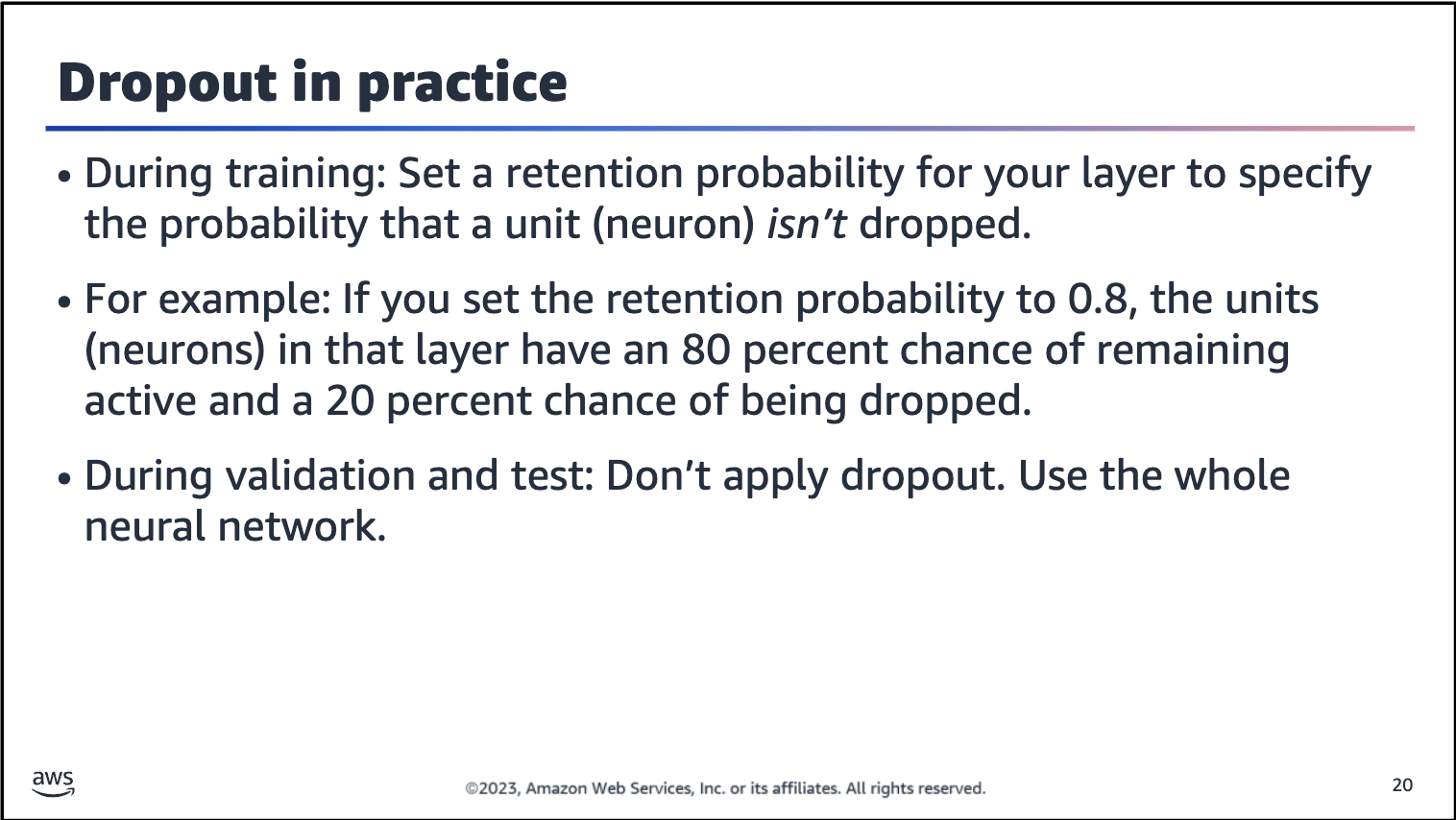


Image description: Diagram of a multilayer network before and after dropout. Before dropout, the network has an input layer with four features, a hidden layer with five neurons, and an output layer with three neurons. After dropout, the input and output layers are the same, but the hidden layer has only three neurons.

See the contrast between the two networks that are shown on the slide. When dropout is used, some neurons are randomly removed.



To apply dropout, you need to set a retention probability for each layer. The retention probability specifies the probability that a unit is not dropped. For example, if you set the retention probability to 0.8, the units in that layer have an 80 percent chance of remaining active and a 20 percent chance of being dropped.

Dropout is only used during training to make the network more robust to fluctuations in the training data. At test time, however, you want to use the full network. You shouldn’t use dropout with the test data or during inference in production because, as you disable neurons randomly, your network will have different outputs every (sequences of) activation. This undermines consistency.



