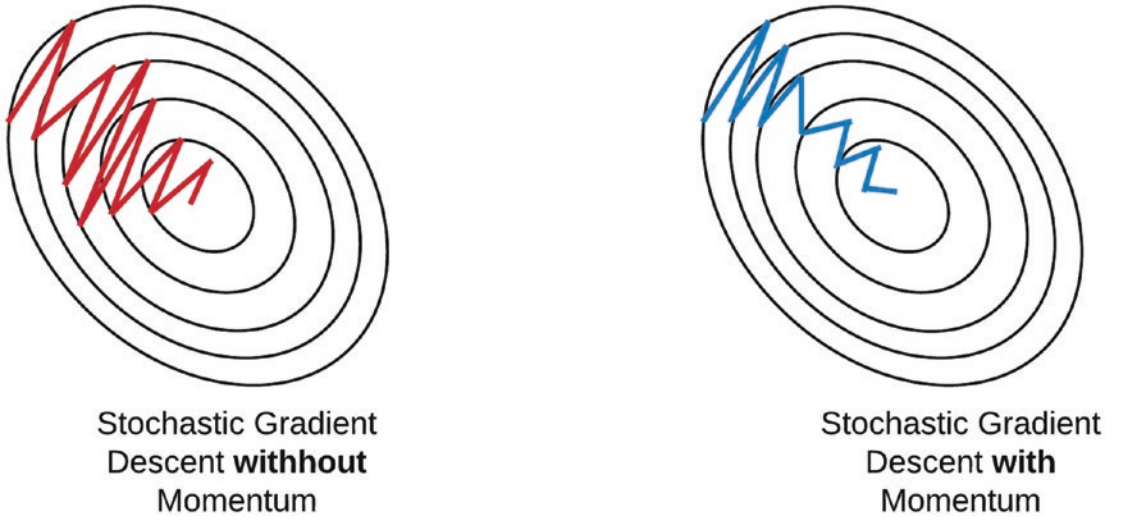

Figure 33-1. *SGD with and without momentum*

Variable Learning Rates

Remember that the learning rate controls how large a step the gradient descent algorithm makes when moving in the direction of steepest descent. If the learning rate is large, the algorithm takes larger steps in the direction of the steepest gradient, as is faster. However, the algorithm may overshoot the global minimum and fail to converge. But if the learning rate is set to a small number, closer to zero, the algorithm converges slowly, but it is more guaranteed to converge.

Variable learning rates are a set of techniques for adjusting the learning rate of the gradient descent algorithm at every time instance while training. These methods are also called learning rate scheduling. Examples of variable learning rates include

- Step decay: This method reduces the learning rate by a constant factor after a certain number of iterations.
- Exponential decay: The exponential decay adapts the learning rate following an exponential distribution.
- Decay proportion: This method reduces the learning rate by a ratio of 1 over the time instance, **t**. The learning rate decay can be adjusted by modifying the proportionality constant.

In TensorFlow 2.0, the **'decay'** parameter of the selected optimizer from the **'tf.keras.optimizers'** module allows time inverse decay of learning rate.

Adaptive Learning Rates

Adaptive learning rate, on the other hand, re-adjusts the learning rate in accordance with the training data. It basically uses a different learning rate for each parameter and adapts it during training. These techniques are based on the observation that each parameter results in a different type of gradient. The following list outlines types of adaptive learning rates in use and their method calls in TensorFlow 2.0:

- AdaGrad: **tf.keras.optimizers.Adagrad()**
- AdaDelta: **tf.keras.optimizers.Adadelta()**
- RMSProp: **tf.keras.optimizers.RMSprop()**
- Adaptive Moments, (Adam): **tf.keras.optimizers.Adam()**

However, AdaGrad performs poorly when used for training deep learning models due to its monotonic learning rate which could be too aggressive, and learning may stop early during training. As of now, there is no proven best optimization technique, so the choice of the optimization technique is down to the preference of the model designer.

This chapter surveys some other techniques for optimizing the weights of a deep neural network. These techniques have implementations in deep learning libraries such as Tensorflow and Keras and can be explored as hyper-parameters when designing a neural network solution for a particular learning use case.

In the next chapter, we will discuss techniques for applying regularization to a deep neural network to prevent overfitting.