Gradient-based optimization (!important!)

- Recall that `output = relu(dot(W, input) + b)`
  - Here, `W` and `b` are tensors called the *weights* or *trainable parameters* of the layer (the `kernel` and `bias` attributes, respectively). `W(k, d)`, input `X(d)`, `b(k)`
  - These weights contain the information learned by the network from exposure of training data.
- Initially, these weight tensors are filled with small random values, a step called *random initialization*. --> This will yield meaningless representations.
- These weights will be gradually adjusted based on a feedback signal.
- This gradual adjustment, also called *training*, is basically the learning that machine learning is all about.
- In a *training loop*, we will do the following things, and repeat these steps as long as necessary:
  - Step 1: Draw a batch of training samples `x` and corresponding targets `y_true`.
  - Step 2: Run the network on `x` (a step called the *forward pass*) to obtain predictions `y_pred`.
  - Step 3: Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y_true`.
  - Step 4: Update all weights of the network in a way that slightly reduces the loss on this batch.
- We'll eventually end up with a network that has a very low loss on its training data: a low mismatch between predictions `y_pred` and expected target `y_true`.
- Step 1, 2, and 3 are straightforward. The only difficult part is Step 4:
  - Given an individual weight coefficient in the network, how can we compute whether the coefficient should be increased or decreased, and by how much?
  - One naive solution: freeze all weights in the model except the one scalar coefficient being considered, and try different values for this coefficient.
- Here comes the *gradient descent*.
- Note that all operations used in the network are *differentiable*!. Therefore, we can compute the *gradient* of the loss with regard to the network's coefficients.
  - Then, we can move the coefficients in the opposite direction from the gradient, thus decreasing the loss.

> ## What is a derivative?

- Consider a continuous and smooth function `f(x) = y`.

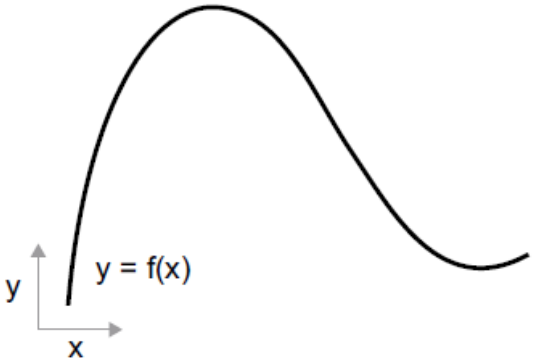


Figure 2.15 A continuous, smooth function

  - Because the function is *continuous*, a small change in `x` can only result in a small change in `y`.
- If you increase `x` by a small factor `epsilon_x`, this results in a small `epsilon_y` change to `y`:
  - `f(x + epsilon_x) = y + epsilon_y`

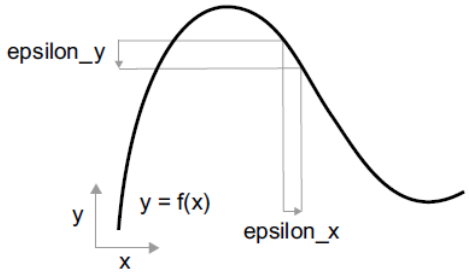


Figure 2.16 With a continuous function, a small change in `x` results in a small change in `y`.

- In addition, because the function is *smooth*, when `epsilon_x` is small enough, around a certain point `p`, it is possible to approximate `f` as a linear function of slope `a`, so that `epsilon_y` becomes `a*epsilon_x`:
  - `f(x + epsilon_x) = y + a*epsilon_x`
  - Obviously, this linear approximation is valid only when `x` is close enough to `p`.
- The slope `a` is called the *derivative* of `f` in `p`.
  - If `a` is negative: a small change of `x` around `p` --> a decrease of `f(x)`.
  - If `a` is positive: a small change of `x` around `p` --> an increase of `f(x)`.
  - The absolute value of `a` (the *magnitude* of the derivative): how quickly this increase or decrease will happen.

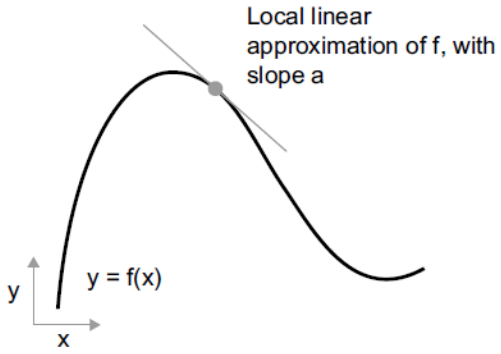


Figure 2.17 Derivative of `f` in `p`

- For every differentiable function `f(x)`, there exists a derivative function `f'(x)` that maps values of `x` to the slope of the local linear approximation of `f` in those points.
- If we know the derivative of `f`, then we just need to move `x` a little in the opposite direction from the derivative to reduce the value of `f(x)`.

Derivative of a tensor operation: the gradient

- A *gradient* is the derivative of a tensor operation.
  - It is the generalization of the concept of derivatives to functions of multidimensional inputs (i.e., tensors).
  - Similar to the local slope of the curve of the function, the gradient of a tensor function represents the *curvature* of the multidimensional surface described by the function.
- Consider an input vector `x`, a matrix `W`, a target `y_true`, and a loss function `loss`.
  - We can use `W` to compute a target candidate `y_pred`, and compute the loss between `y_pred` and `y_true`. For example,
    - `y_pred = dot(W, x)`
    - `loss_value = loss(y_pred, y_true)`
  - Since `x` and `y_true` are fixed inputs, this can be interpreted as a function mapping values of `W` to loss values:
    - `loss_value = f(W)`
- Suppose that the current value of `W` is `W0`.
  - Then, the derivative of `f` in the point `W0` is a tensor `grad(loss_value, W0)` with the same shape as `W`.
  - Here, each coefficient `grad(loss_value, W0)[i,j]` indicates the direction and magnitude of the change in `loss_value` when modifying `W0[i,j]`.
  - The tensor `grad(loss_value, W0)` is the gradient of the function `f(W) = loss_value` in `W0`. = the gradient of `loss_value` with respect to `W` around `W0`

  - We know that the derivative of a function `f(x)` of a single coefficient can be interpreted as the slope of the curve of `f`.
  - Likewise, `grad(loss_value, W0)` can be interpreted as the tensor describing the *direction of steepest ascent* of `f(W)=loss_value` around `W0`, as well as the slope of this ascent.
  - Each partial derivative describes the slope of `f` in a specific direction.
- Therefore, you can reduce `f(W)` by moving `W` in the opposite direction from the gradient:

  - For example, `W1 = W0 - step*grad(loss_value, W0)` where `step` is a small constant.
  - Note that `step` is needed because `grad(loss_value, W0)` only approximates the slope where we are close to `W0`.

Stochastic gradient descent

- We are dealing with a differentiable function, so we can compute its gradient.
- It means that if we update the weights in the opposite direction from the gradient, the loss will be a little less every time.
- Now, our training loop will consist of the following steps:

  - Step 1: Draw a batch of training samples `x` and corresponding targets `y_true`.
  - Step 2: Run the network on `x` to obtain predictions `y_pred` (called the *forward pass*).
  - Step 3: Compute the loss of the network on the batch, a measure of the mismatch between `y_pred` and `y_true`.
  - Step 4: Compute the gradient of the loss with regard to the network's parameters (called the *backward pass*)
  - Step 5: Move the parameters a little in the opposite direction from the gradient - for example, `W -= learning_rate*gradient` - thus reducing the loss on the batch a bit. The `learning_rate` is a scalar factor modulating the "speed" of the gradient descent process.
- The above procedure for updating the weights is called *mini-batch stochastic gradient descent* (mini-batch SGD).

  - The term *stochastic*: each batch of data is drawn at random.
  - The below figure illustrates what happens in 1D, when the network has only one parameter and we have only one training sample.

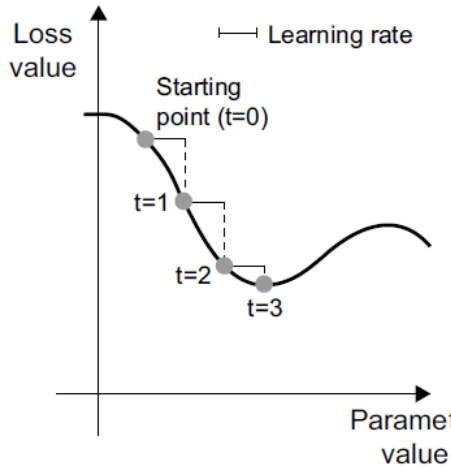


Figure 2.18 SGD down a 1D loss curve (one learnable parameter)

- As we can see, it is important to have a reasonable value for the `learning_rate` parameter.
  - If it's too small, we need many iterations and it could get stuck in a local minimum.
  - It it's too large, we may have completely random locations on the curve.

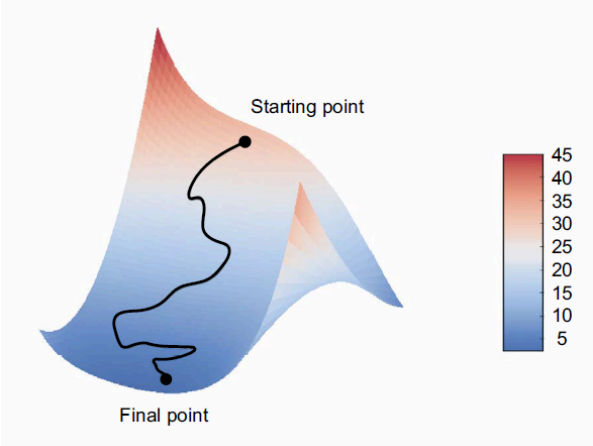


Figure 2.19 Gradient descent down a 2D loss surface (two learnable parameters)

- *(True) SGD* VS *mini-batch SGD* VS *batch GD*
- There exist multiple variants of SGD that differ by taking into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients.
  - SGD with momentum, AdaGrad, RMSProp, etc.
  - Such variants are known as *optimization methods* or simply *optimizers*.
- The concept of *momentum*
  - It addresses two issues with SGD: convergence speed and local minima

- Local minimum and global minimum

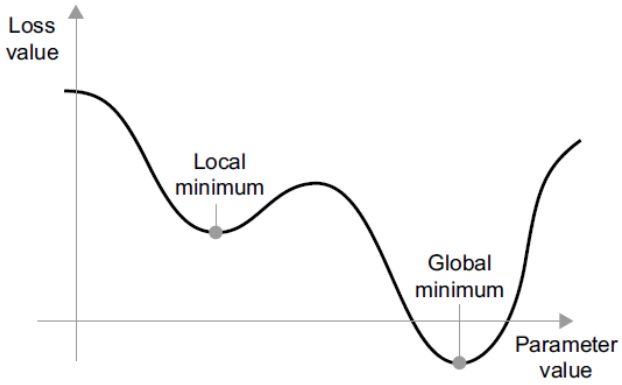


Figure 2.20 A local minimum and a global minimum

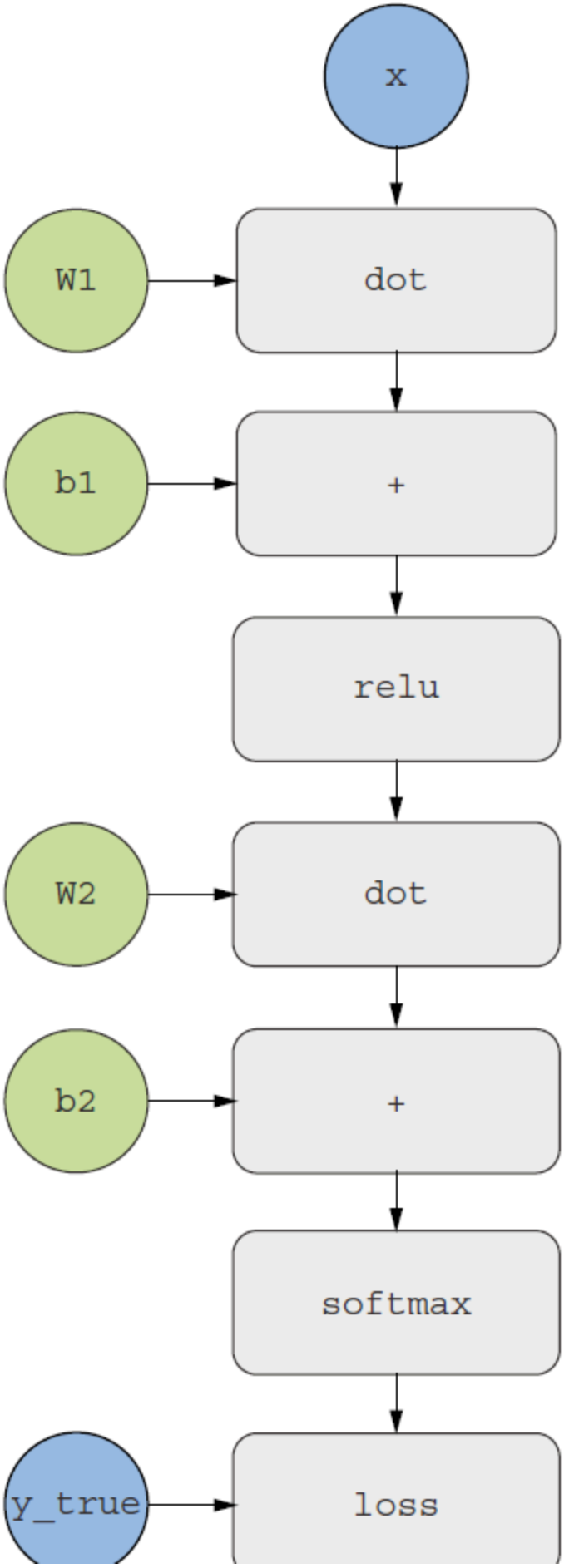
- We can avoid such local minimums by using momentum.
- Think of the optimization process as a small ball rolling down the loss curve.
- It is implemented by moving the ball at each step based not only the current slope value but also on the current velocity.
- In practice, it means updating the parameter `W` based not only the current gradient value but also on the previous parameter update:

```
past_velocity = 0.
momentum = 0.1
while loss > 0.01:
    w, loss, gradient = get_current_parameters()
    velocity = past_velocity * momentum + learning_rate * gradient
    w = w + momentum * velocity - learning_rate * gradient
    past_velocity = velocity
    update_parameter(w)
```

Chaining derivatives: the Backpropagation algorithm

- How can we compute the gradient of complex expressions in practice? In the two-layer model we started this section with, how can we get the gradient of the loss with respect to the weights?
- That's where the *Backpropagation algorithm* comes in.
  - Backpropgation is a way to use the derivatives of simple operations (such as addition, relu, or tensor product) to easily compute the gradient of arbitrarily complex combinations of these atomic operations.
- The chain rule
  - Note that a neural network function consists of many tensor operations chained together, each of which has a simple, known derivative.

    - For example, consider a network `f` composed of three tensor operations, `a`, `b`, and `c`, with weight matrices `W1`, `W2`, and `W3`:
    - `f(W1, W2, W3) = a(W1, b(W2, c(W3)))`
    - Such a chain of functions can be derived using the following *chain rule*:
        - The derivative of `f(g(x)) = f'(g(x)) * g'(x)`
- Automatic differentiation with computation graphs
    - Consider `loss_value = loss(y_true, softmax(dot(relu(dot(inputs, W1) + b1), W2) + b2))`.
    - The computation graph representation


x
W1
dot
b1
+
relu
W2
dot
b2
+
softmax
y_true
loss

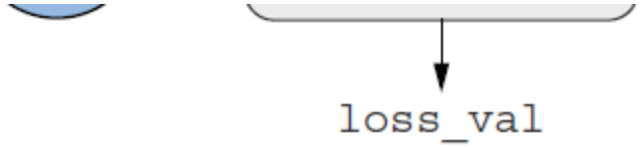


Figure 2.21 The computation graph representation of our two-layer model

- To explain backpropagation clearly, let's look at a really basic example of a computation graph: only one linear layer and where all variables are scalar.
  - Two scalar variables `w` and `b`, a scalar input `x`, and an absolute error-loss function `loss_val = abs(y_true - y)`
  - We are interested in computing `grad(loss_val, b)` and `grad(loss_val, w)`.

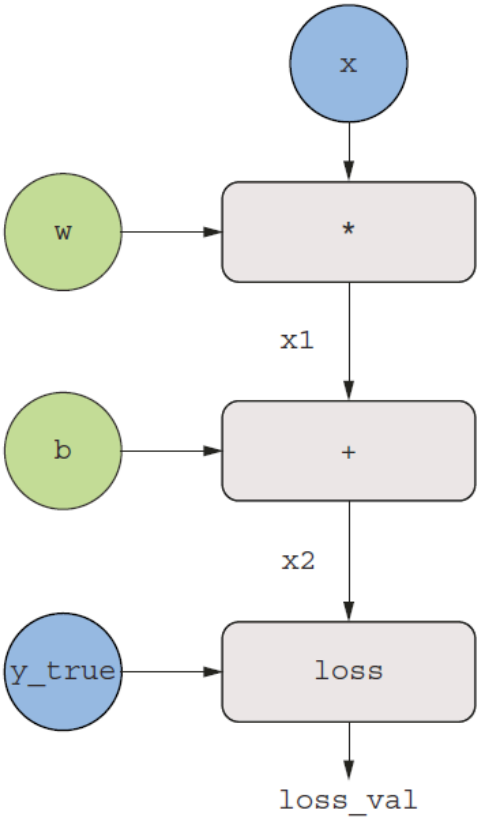


Figure 2.22 A basic example of a computation graph

- The forward pass

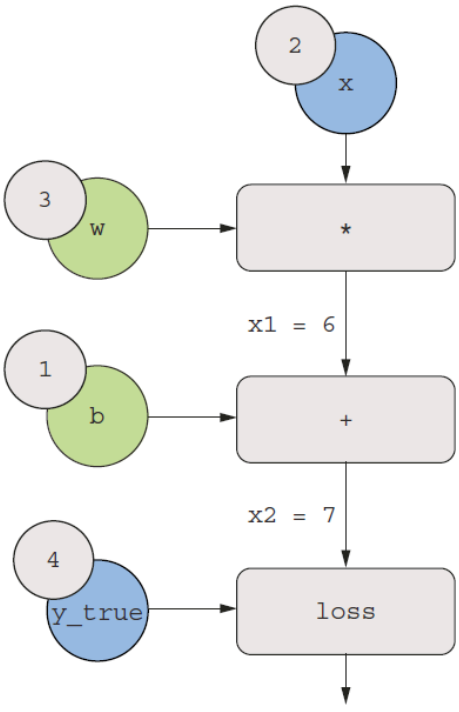


Figure 2.23 Running a forward pass

- Let's *reverse* the graph: this backward graph represents the *backward* pass.

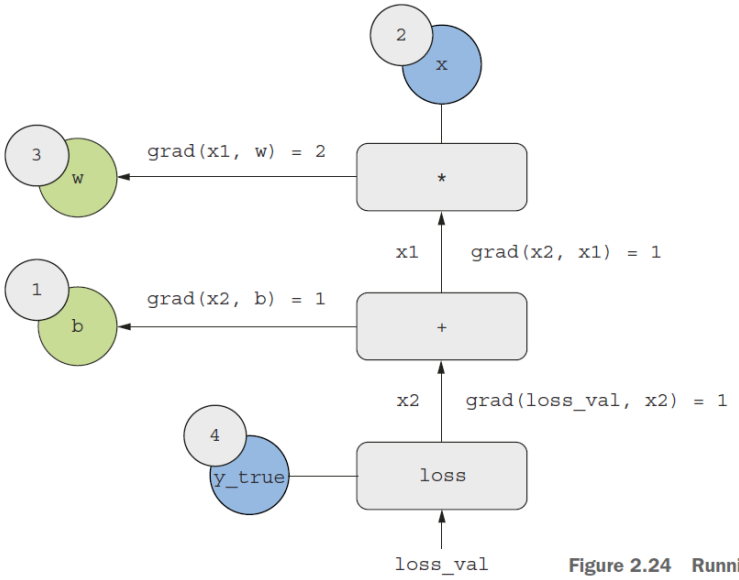


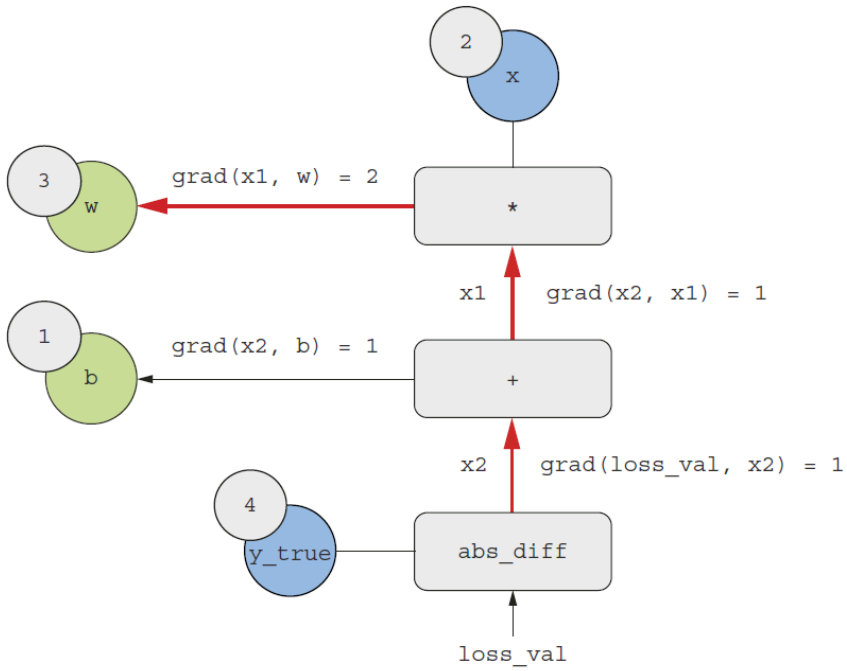
Figure 2.24 Running a backward pass

- We have the following:
  - `grad(loss_val, x2) = 1` since as `x2` varies by an amount epsilon, `loss_val = abs(4 - x2)` varies by the same amount.
  - `grad(x2, x1) = 1` since as `x1` varies by an amount epsilon, `x2 = x1 + b = x1 + 1` varies by the same amount.
  - `grad(x2, b) = 1` since as `b` varies by an amount epsilon, `x2 = x1 + b = 6 + b` varies by the same amount.

- `grad(x1, w) = 2` since as `w` varies by an amount epsilon, `x1 = x * w = 2 * w` varies by `2 * epsilon`.

- The chain rule on this backward graph says "you can obtain the derivative of a node with respect to another node by *multiplying the derivatives for each edge along with the path linking the two nodes*".

  - For instance, `grad(loss_val, w) = grad(loss_val, x2) * grad(x2, x1) * grad(x1, w)`



  - By applying the chain rule to this graph, we obtain what we were looking for:

    - `grad(loss_val, w) = 1 * 1 * 2 = 2`
    - `grad(loss_val, b) = 1 * 1 = 2`

- Note that if there are multiple paths linking the two nodes of interest, `a` and `b`, in the backward graph, we would obtain `grad(b, a)` by summing the contributions of all the paths.

- Backpropagation is simply the application of the chain rule to a computation graph.

  - It starts with the final loss value and works backward from the top layer to the bottom layers.
  - We "back propagate" the loss contributions of different nodes in a computation graph.

- Automatic differentiation

  - Modern frameworks are capable of automatic differentiation.

    - Automatic differentiation makes it possible to retrieve the gradients of arbitrary compositions of differentiable tensor operations without doing any extra work besides writing down the forward pass.

Summary

- *Tensors* form the foundation of modern machine learning systems. They come in various flavors of `dtype`, `rank`, and `shape`.
- You can manipulate numerical tensors via `tensor operations` (such as addition, tensor product, or element-wise multiplication), which can be interpreted as encoding geometric transformations. In general, everything in deep learning is amenable to a geometric interpretation.
- Deep learning models consist of chains of simple tensor operations, parameterized by *weights*, which are themselves tensors. The weights of a model are where its "knowledge" is stored.
- *Learning* means finding a combination of model parameters that minimizes a loss function for a given set of training data samples and their corresponding targets.
- Learning happens by drawing random batches of data samples and their targets, and computing the gradient of the network parameters with respect to the loss on the batch. The network parameters are then moved a bit (the magnitude of the move is defined by the learning rate) in the opposite direction from the gradient. This is called *mini-batch stochastic gradient descent*.
- The entire learning process is made possible by the fact that neural networks are chains of differentiable tensor operations, and thus it is possible to apply the chain rule of derivation to find the gradient function mapping the current parameters and current batch of data to a gradient value. This is called *backpropagation*.
- Two key concepts you'll see frequently in future chapters are *loss* and *optimizers*. These are the two things you need to define before you begin feeding data into a network.
    - The *loss* in the quantity you'll attempt to minimize during training, so it should represent a measure of success for the task you're trying to solve.
    - The *optimizer* specifies the exact way in which the gradient of the loss will be used to update parameters: for instance, it could be the RMSProp optimizer, SGD with momentum, and so on.

Reading assignments

- Section 2.4 and 2.5 in "Dive into deep learning"