The Mathematical Building Blocks of Neural Networks

Data Representations for Neural Networks: Tensor

- All current deep learning systems use tensors as their basic data structure.
- Tensors are fundamental to the field—so fundamental that TensorFlow was named after them.
- A tensor is a container for data—usually numerical data.

Tensors

- Scalars (rank-0 tensors)
  - A tensor that contains only one number is called a scalar (or scalar tensor, or rank-0 tensor, or 0D tensor).
  - In NumPy, a `float32` or `float64` number is a scalar tensor (or scalar array).

```
>>> import numpy as np
>>> x = np.array(12)
>>> x
array(12)
>>> x.ndim
0
```

  - `ndim` attribute displays the number of axes of a NumPy tensor
  - A scalar tensor has 0 axes `(ndim == 0)`.
  - The number of axes of a tensor is also called its *rank*.

Tensors…

- Vectors (rank-1 tensors)
  - An array of numbers is called a *vector*, or rank-1 tensor, or 1D tensor.
  - A rank-1 tensor has exactly one axis.

```
>>> x = np.array([12, 3, 6, 14, 7])
>>> x
array([12, 3, 6, 14, 7])
>>> x.ndim
1
```

  - This vector has five entries and so is called a 5-dimensional vector.
  - Don't confuse a 5D vector with a 5D tensor!
  - A 5D vector has only one axis and has five dimensions along its axis, whereas a 5D tensor has five axes (and may have any number of dimensions along each axis).

Tensors…

- *Dimensionality* can denote either the number of entries along a specific axis (as in the case of our 5D vector) or the number of axes in a tensor (such as a 5D tensor), which can be confusing at times.
- In the latter case, it's technically more correct to talk about a tensor of rank 5 (the rank of a tensor being the number of axes), but the ambiguous notation *5D tensor* is common regardless.

Tensors…

- Matrices (rank-2 tensors)
  - An array of vectors is a *matrix*, or rank-2 tensor, or 2D tensor.
  - A matrix has two axes (often referred to as *rows* and *columns*).
  - You can visually interpret a matrix as a rectangular grid of numbers.

```
>>> x = np.array([[5, 78, 2, 34, 0],
                  [6, 79, 3, 35, 1],
                  [7, 80, 4, 36, 2]])
>>> x.ndim 2
```

Tensors…

- Rank-3 tensors
  - If you pack such matrices in a new array, you obtain a rank-3 tensor (or 3D tensor), which you can visually interpret as a cube of numbers.

```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]]])
>>> x.ndim
3
```

Tensors…

- Higher-rank tensors
  - By packing rank-3 tensors in an array, you can create a rank-4 tensor, and so on.
  - In deep learning, you'll generally manipulate tensors with ranks 0 to 4, although you may go up to 5 if you process video data.

Tensors: Key Attributes

- A tensor is defined by three key attributes:
  - Number of axes (rank) - for instance, a rank-3 tensor has three axes, and a matrix has two axes.
  - Shape – tuple of integers that describes how many dimensions the tensor has along each axis.
    - The previous matrix example has shape (3, 5)
    - The rank-3 tensor example has shape (3, 3, 5)
    - A vector has a shape with a single element, such as (5,)
    - A scalar has an empty shape, ()
  - Data type (usually called `dtype` in Python libraries)—
    - Type of the data contained in the tensor
    - For instance, a tensor's type could be `float16`, `float32`, `float64`, `uint8`, and so on.

Manipulating Tensors in NumPy

- Selecting specific elements in a tensor is called tensor slicing.

```
>>> train_images.shape
(60000, 28, 28)
>>> my_slice = train_images[10:100]
>>> my_slice.shape
(90, 28, 28)
```

or,

```
>>> my_slice = train_images[10:100, :, :]
>>> my_slice.shape
(90, 28, 28)
```

or,

```
>>> my_slice = train_images[10:100, 0:28, 0:28]
>>> my_slice.shape
(90, 28, 28)
```

Manipulating Tensors in NumPy

- Selects bottom-right corner of all images

```
my_slice = train_images[:, 14:, 14:]
```

- Selects patches of 14 × 14 pixels centered in the middle (uses negative index)

```
my_slice = train_images[:, 7:-7, 7:-7]
```

Notion of Data Batches

- In general, the first axis (axis 0) in all data tensors in deep learning will be the samples axis (or samples dimension).
- Deep learning models don't process an entire dataset at once; rather, they break the data into small batches.
  - For example, one batch of MNIST digits, with a batch size of 128:

```
batch = train_images[:128]
```

  - And here's the next batch: `batch = train_images[128:256]`
  - And the nth batch:

```
batch = train_images[128 * n:128 * (n + 1)]
```

- When considering such a batch tensor, the first axis (axis 0) is called the batch axis or batch dimension.

Real-world Examples of Data Tensors

- Vector data
  - Rank-2 tensors of shape `(samples, features)`
  - each sample is a vector of numerical attributes (“features”)
- Timeseries data or sequence data
  - Rank-3 tensors of shape `(samples, timesteps, features)`
  - each sample is a sequence (of length `timesteps`) of feature vectors
- Images
  - Rank-4 tensors of shape `(samples, height, width, channels)`
  - each sample is a 2D grid of pixels, and each pixel is a vector of values (“channels”)
- Video
  - Rank-5 tensors of shape `(samples, frames, height, width, channels)`
  - each sample is a sequence (of length `frames`) of images

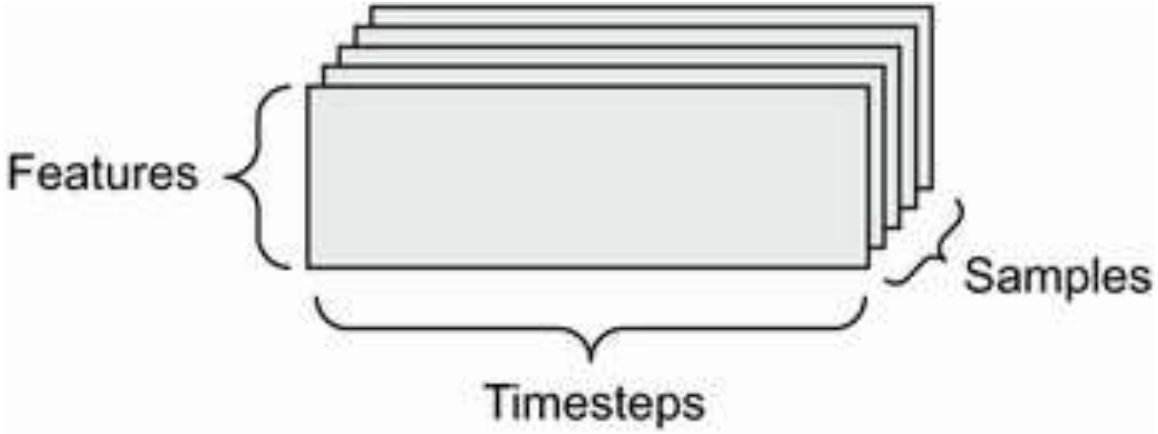
Vector Data

- Each single data point can be encoded as a vector.
- Thus a batch of data will be encoded as a rank-2 tensor (that is, an array of vectors), where the first axis is the samples axis and the second axis is the features axis.

Vector Data: Examples

- An actuarial dataset of people,
  - where we consider each person's age, gender, and income.
  - Each person can be characterized as a vector of 3 values
  - Thus an entire dataset of 100,000 people can be stored in a rank-2 tensor of shape `(100000, 3)`.
- A dataset of text documents,
  - where we represent each document by the counts of how many times each word appears in it (out of a dictionary of 20,000 common words).
  - Each document can be encoded as a vector of 20,000 values (one count per word in the dictionary), and thus an entire dataset of 500 documents can be stored in a tensor of shape `(500, 20000)`.

Timeseries or Sequence Data



A rank-3 timeseries data tensor

- Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a rank-3 tensor with an explicit time axis.
- Each sample can be encoded as a sequence of vectors (a rank-2 tensor), and thus a batch of data will be encoded as a rank-3 tensor.

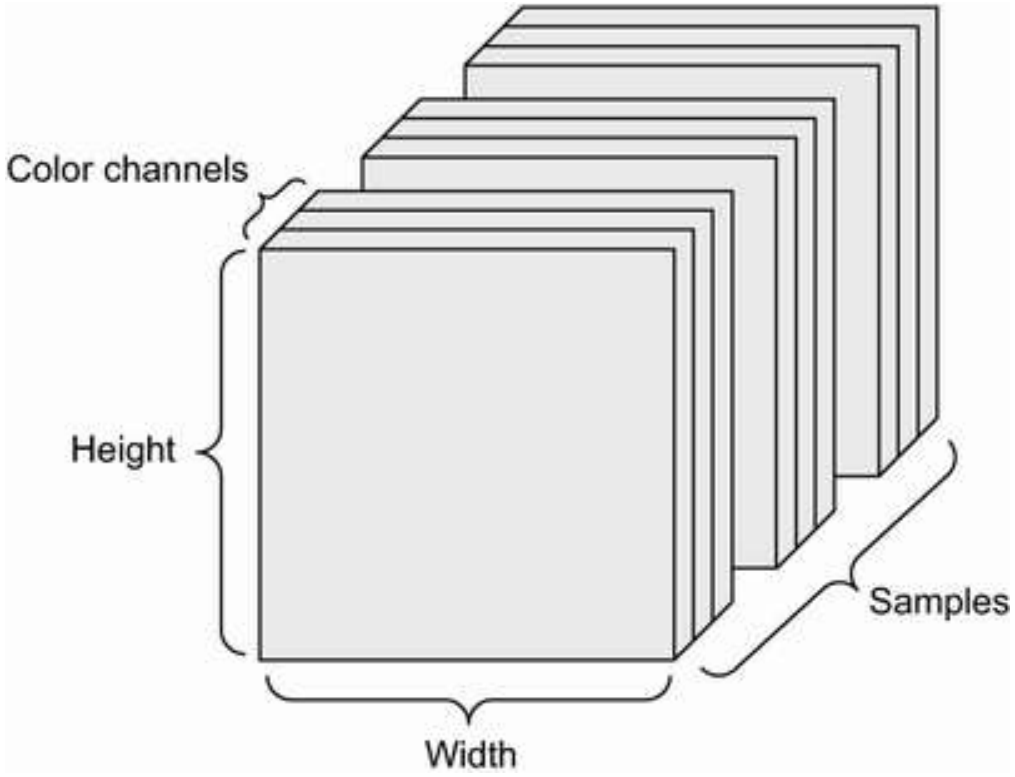
Timeseries or Sequence Data: Examples

- A dataset of stock prices
  - Every minute, we store the current price of the stock, the highest price in the past minute, and the lowest price in the past minute.
  - Thus, every minute is encoded as a 3D vector, an entire day of trading is encoded as a matrix of shape `(390, 3)` (there are 390 minutes in a trading day), and 250 days' worth of data can be stored in a rank-3 tensor of shape `(250, 390, 3)`.
  - Here, each sample would be one day's worth of data.

Timeseries or Sequence Data: Examples

- A dataset of tweets,
  - where we encode each tweet as a sequence of 280 characters out of an alphabet of 128 unique characters.
  - In this setting, each character can be encoded as a binary vector of size 128 (an all-zeros vector except for a 1 entry at the index corresponding to the character).
  - Then each tweet can be encoded as a rank-2 tensor of shape `(280, 128)`, and a dataset of 1 million tweets can be stored in a tensor of shape `(1000000, 280, 128)`.

Image Data



A rank-4 image data tensor

- Images typically have three dimensions: height, width, and color depth.
- Although grayscale images have only a single color channel and could thus be stored in rank-2 tensors, by convention image tensors are always rank-3, with a one-dimensional color channel for grayscale images.
- A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape `(128, 256, 256, 1)`, and a batch of 128 color images could be stored in a tensor of shape `(128, 256, 256, 3)`.

Video Data

- Video data is one of the few types of real-world data for which you'll need rank-5 tensors.
- A video can be understood as a sequence of frames, each frame being a color image.
- Because each frame can be stored in a rank-3 tensor `(height, width, color_ depth)`, a sequence of frames can be stored in a rank-4 tensor `(frames, height, width, color_depth)`.
- Thus a batch of different videos can be stored in a rank-5 tensor of shape `(samples, frames, height, width, color_depth)`.

Video Data

- For instance, a 60-second, 144 × 256 YouTube video clip sampled at 4 frames per second would have 240 frames.
- A batch of four such video clips would be stored in a tensor of shape `(4, 240, 144, 256, 3)`.
- That's a total of 106,168,320 values!
- If the `dtype` of the tensor was `float32`, each value would be stored in 32 bits, so the tensor would represent 405 MB. Heavy!
- Videos you encounter in real life are much lighter, because they aren't stored in `float32`, and they're typically compressed by a large factor (such as in the MPEG format).

The Gears of Neural Networks: Tensor operations

The Gears of Neural Networks: Tensor operations

- Any computer program can be ultimately reduced to a small set of binary operations on binary inputs (AND, OR, NOR, and so on).
- All transformations learned by deep neural networks can be reduced to a handful of tensor operations (or tensor functions) applied to tensors of numeric data.
- For instance, it's possible to add tensors, multiply tensors, and so on.

The Gears of Neural Networks: Tensor operations

- We build our model by stacking layers on top of each other.
- This layers can be interpreted as a function, which takes as input a matrix and returns another matrix—a new representation for the input tensor.
- Specifically, the function is as follows (where `W` is a matrix and `b` is a vector, both attributes of the layer):

```
output = relu(dot(input, W) + b)
```

The Gears of Neural Networks: Tensor operations

```
output = relu(dot(input, W) + b)
```

- We have three tensor operations here:
  - A dot product (`dot`) between the input tensor and a tensor named `W`
  - An addition (`+`) between the resulting matrix and a vector `b`
  - A `relu` operation:
    - `relu(x)` is `max(x, 0)`;
    - "relu" stands for "rectified linear unit"

Element-wise Operations

- The `relu` operation and addition are element-wise operations:
  - operations that are applied independently to each entry in the tensors being considered.
- This means these operations are highly amenable to massively parallel implementations.

```
import numpy as np
z = x + y                   Element-wise addition
z = np.maximum(z, 0.)       Element-wise relu
```

Broadcasting

- What happens with addition when the shapes of the two tensors being operated differ?
  - When possible, and if there's no ambiguity, the smaller tensor will be *broadcast* to match the shape of the larger tensor.
- Broadcasting consists of two steps:
  - Axes (called broadcast axes) are added to the smaller tensor to match the `ndim` of the larger tensor.
  - The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor.

Broadcasting

- Consider X with shape (32, 10) and y with shape (10,):

```
import numpy as np
X = np.random.random((32, 10))          # shape(32, 10)
y = np.random.random((10,))             # shape(10,)
```

- First, we add an empty first axis to y, whose shape becomes (1, 10):

```
y = np.expand_dims(y, axis=0)           # shape(1, 10)
```

- Then, we repeat y 32 times alongside this new axis, so that we end up with a tensor Y with shape (32, 10), where Y[i, :] == y for i in range(0, 32):

```
Y = np.concatenate([y] * 32, axis=0)            # shape(32, 10)
```

Broadcasting

- With broadcasting, you can generally perform element-wise operations that take two inputs tensors if one tensor has shape `(a, b, … n, n + 1, … m)` and the other has shape `(n, n + 1, … m)`.
- The broadcasting will then automatically happen for axes `a` through `n - 1`.

Tensor Product

- The tensor product, or dot product (not to be confused with an element-wise product, the * operator), is one of the most common, most useful tensor operations.

```
x = np.random.random((32,))
y = np.random.random((32,))
z = np.dot(x, y)
```

- In mathematical notation, you'd note the operation with a dot (•):

$$z = x \cdot y$$

Tensor Product

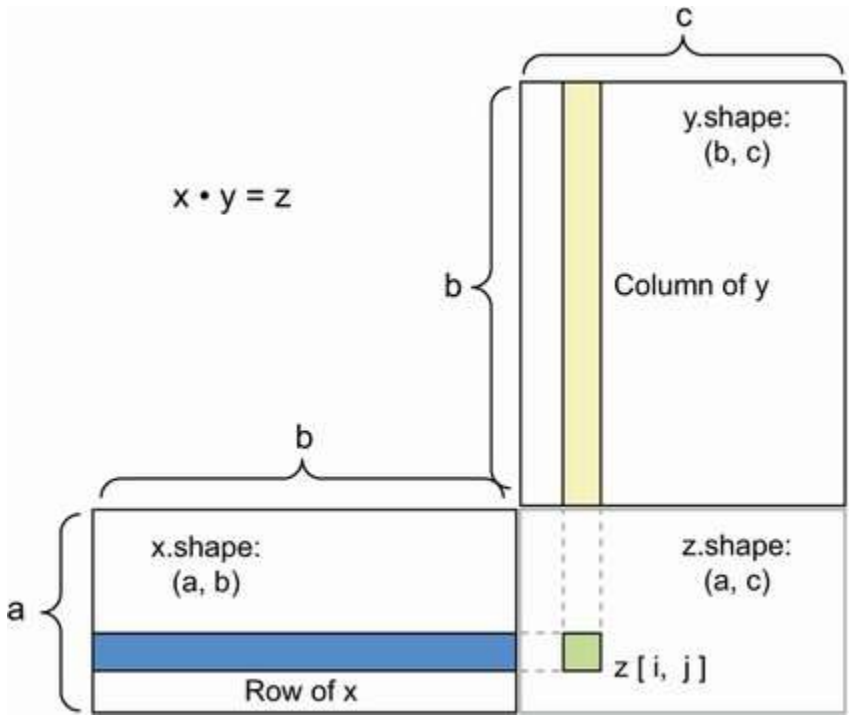
- The dot product between two vectors is a scalar.
- That only vectors with the same number of elements are compatible for a dot product.
- The dot product between a matrix x and a vector y, which returns a vector where the coefficients are the dot products between y and the rows of x.

Tensor Product

- A dot product generalizes to tensors with an arbitrary number of axes.
- The most common applications may be the dot product between two matrices.
- You can take the dot product of two matrices `x` and `y`:

  `(dot(x, y))` if and only if `x.shape[1] == y.shape[0]`
- The result is a matrix with shape `(x.shape[0], y.shape[1])`, where the coefficients are the vector products between the rows of `x` and the columns of `y`.

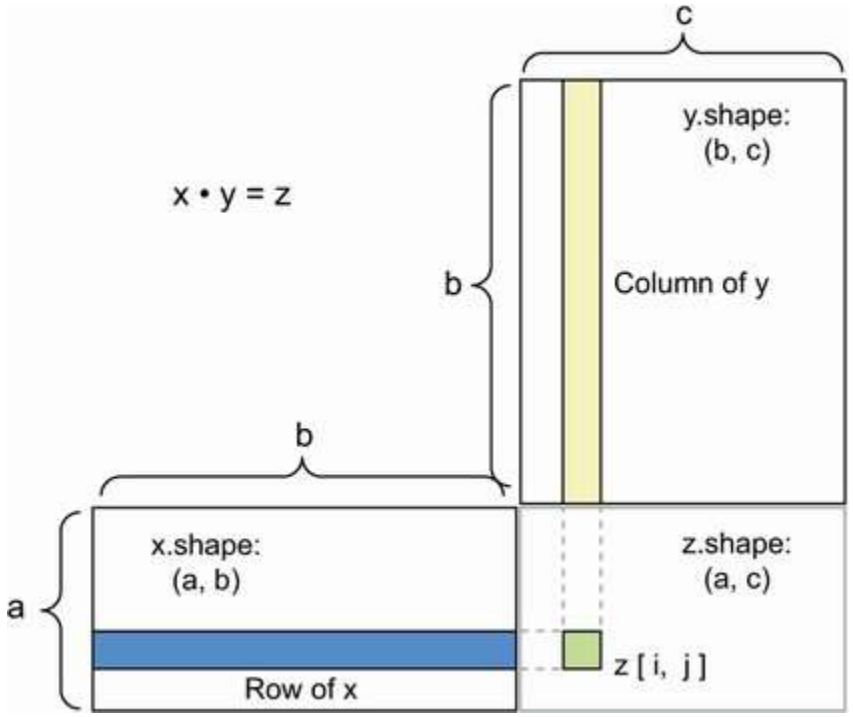
Tensor Product



Matrix dot-product box diagram

- In the figure, `x`, `y`, and `z` are pictured as rectangles (literal boxes of coefficients).
- Because the rows of `x` and the columns of `y` must have the same size, it follows that the width of `x` must match the height of `y`.

Tensor Product



Matrix dot-product box diagram

- More generally, you can take the dot product between higher-dimensional tensors, following the same rules for shape compatibility as outlined earlier for the 2D case:

```
(a, b, c, d) • (d,) → (a, b, c)
(a, b, c, d) • (d, e) → (a, b, c, e)
```

And so on.

Tensor Reshaping

- A third type of tensor operation that's essential to understand is tensor reshaping.
- We use it when we preprocess the digits data before feeding it into our model.

Tensor Reshaping

- Reshaping a tensor means rearranging its rows and columns to match a target shape.
- Naturally, the reshaped tensor has the same total number of coefficients as the initial tensor.

```
>>> x = np.array([[0., 1.],
                  [2., 3.],
                  [4., 5.]])
>>> x.shape
(3, 2)
>>> x = x.reshape((6, 1))
>>> x
array([[ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
```

```
>>> x = x.reshape((2, 3))
>>> x
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

Tensor Reshaping

- A special case of reshaping that's commonly encountered is transposition.
- Transposing a matrix means exchanging its rows and its columns, so that `x[i, :]` becomes `x[:, i]`:

```
>>> x = np.zeros((300, 20))
>>> x = np.transpose(x)
>>> x.shape (20, 300)
```

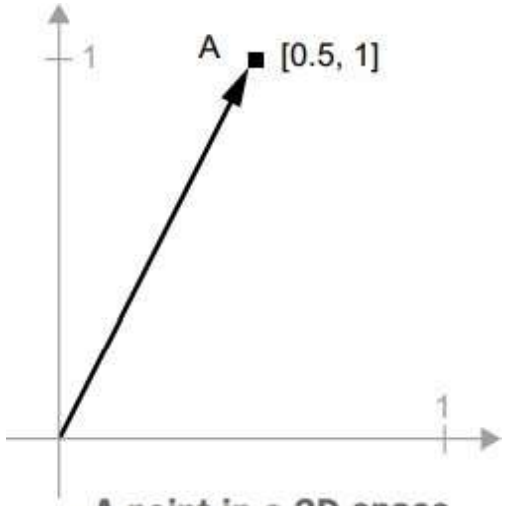
Creates an allzeros matrix of shape (300, 20)

Geometric Interpretation of Tensor Operations

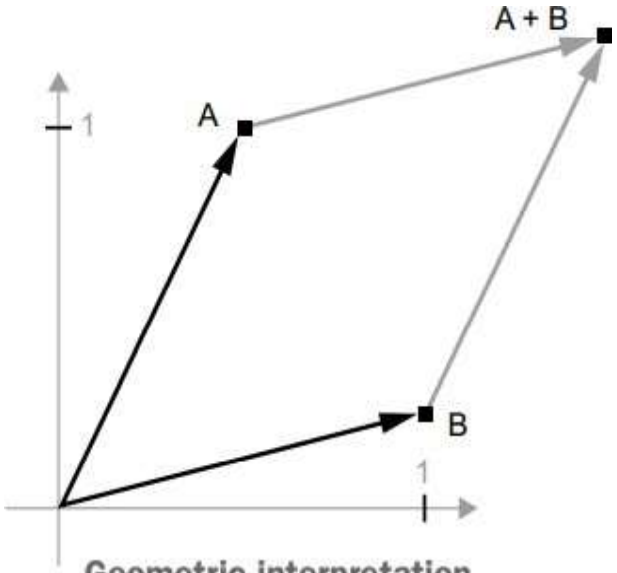
- The contents of the tensors manipulated by tensor operations can be interpreted as coordinates of points in some geometric space.
- Thus all tensors operations have a geometric interpretation.

Geometric Interpretation of Tensor Operations

- Adding a vector B to a vector A represents the action of copying point A in a new location, whose distance and direction from the original point A is determined by the vector B.



A point in a 2D space pictured as an arrow



Geometric interpretation of the sum of two vectors

Geometric Interpretation of Tensor Operations

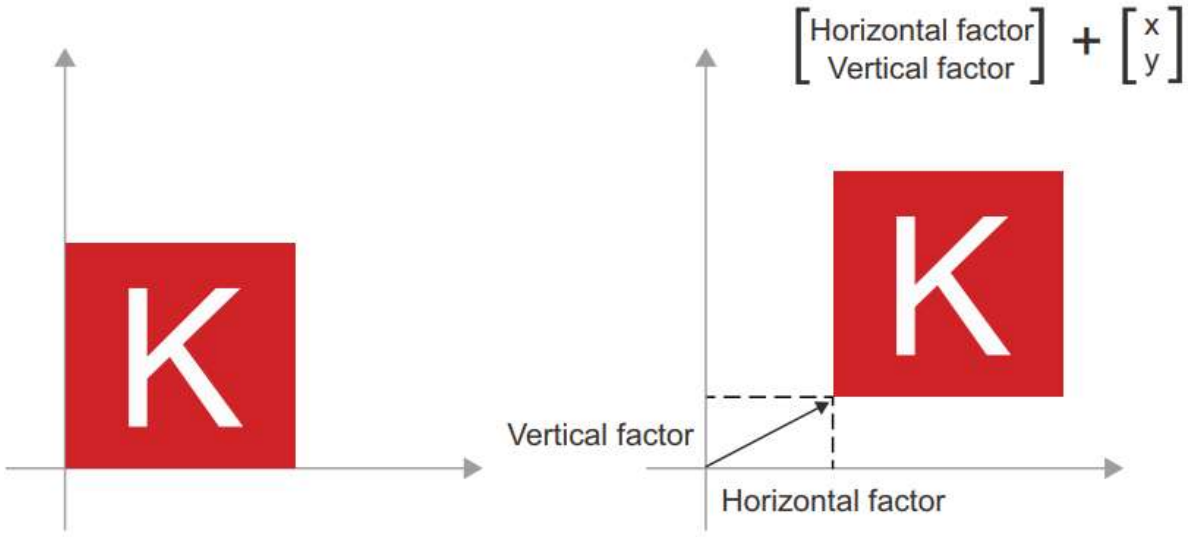
- If you apply the same vector addition to a group of points in the plane (an “object”), you would be creating a copy of the entire object in a new location.
- Tensor addition thus represents the action of translating an object (moving the object without distorting it) by a certain amount in a certain direction.

Geometric Interpretation of Tensor Operations

- In general, elementary geometric operations such as translation, rotation, scaling, skewing, and so on can be expressed as tensor operations.

Translations

- Adding a vector to a point will move the point by a fixed amount in a fixed direction.
- Applied to a set of points (such as a 2D object), this is called a "translation".

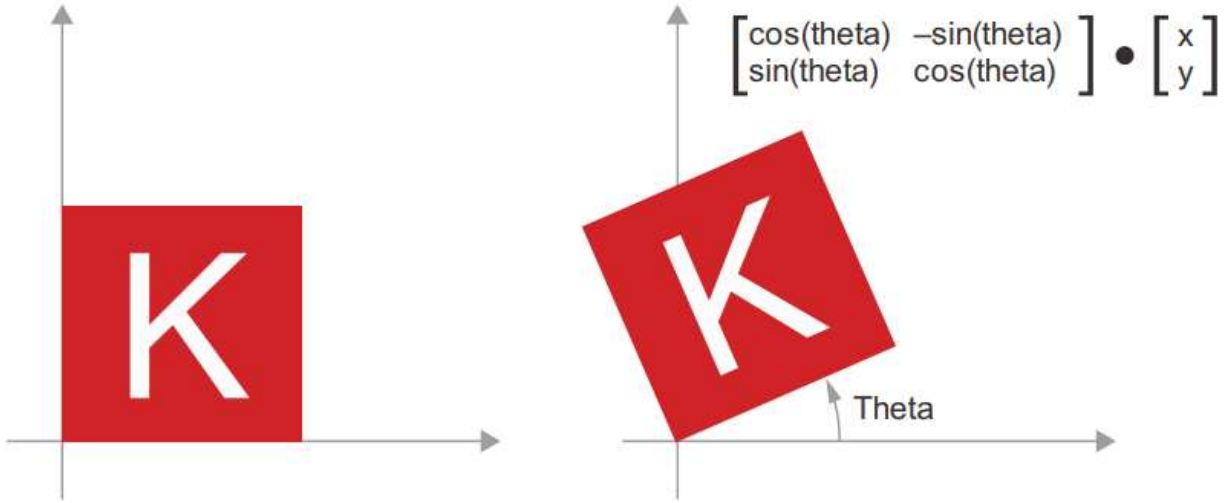


2D translation as a vector addition

Rotation

- A counterclockwise rotation of a 2D vector by an angle theta can be achieved via a dot product with a 2 × 2 matrix

```
R = [[cos(theta), -sin(theta)], [sin(theta), cos(theta)]]
```

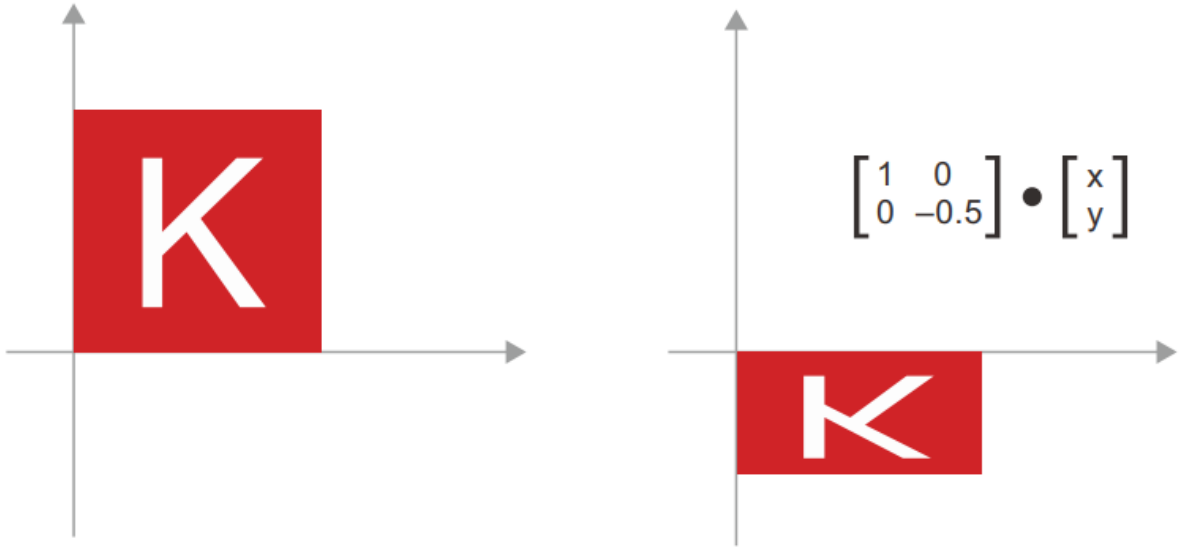


2D rotation (counterclockwise) as a dot product

Scaling

- A vertical and horizontal scaling of the image can be achieved via a dot product with a 2 × 2 diagonal matrix

```
S = [[horizontal_factor, 0], [0, vertical_factor]]
```



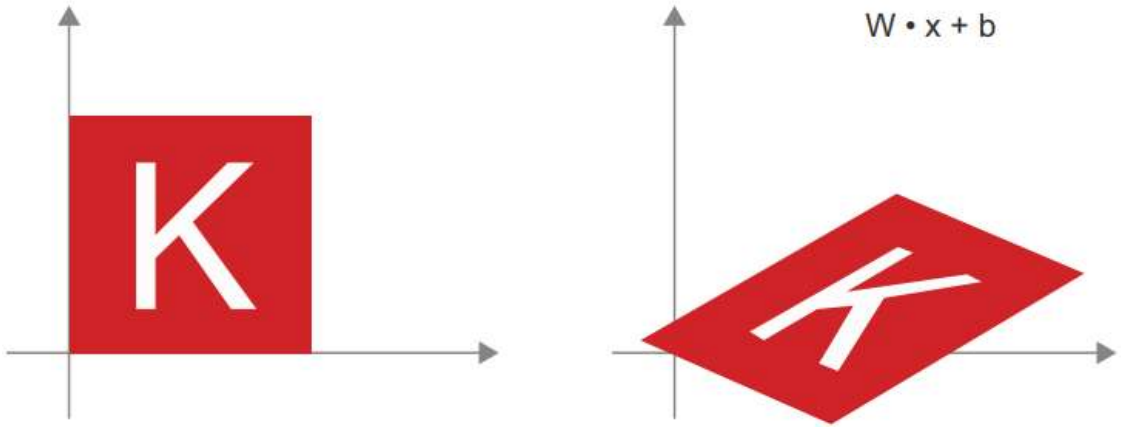
2D scaling as a dot product

Linear Transform

- A dot product with an arbitrary matrix implements a linear transform.
- Note that scaling and rotation are by definition linear transforms.

Affine Transform

- An affine transform is the combination of a linear transform and a translation.
- That's exactly the `y = W • x + b` computation implemented by the `Dense` layer!
- A `Dense` layer without an activation function is an affine layer.



Affine transform in the plane

`Dense` Layer with `relu` Activation

- An important observation about affine transforms is that if you apply many of them repeatedly, you still end up with an affine transform (so you could just have applied that one affine transform in the first place).
- Let's try it with two:

$$\texttt{affine2(affine1(x))} = \texttt{W2} \cdot (\texttt{W1} \cdot \texttt{x} + \texttt{b1}) + \texttt{b2}$$
$$= (\texttt{W2} \cdot \texttt{W1}) \cdot \texttt{x} + (\texttt{W2} \cdot \texttt{b1} + \texttt{b2})$$

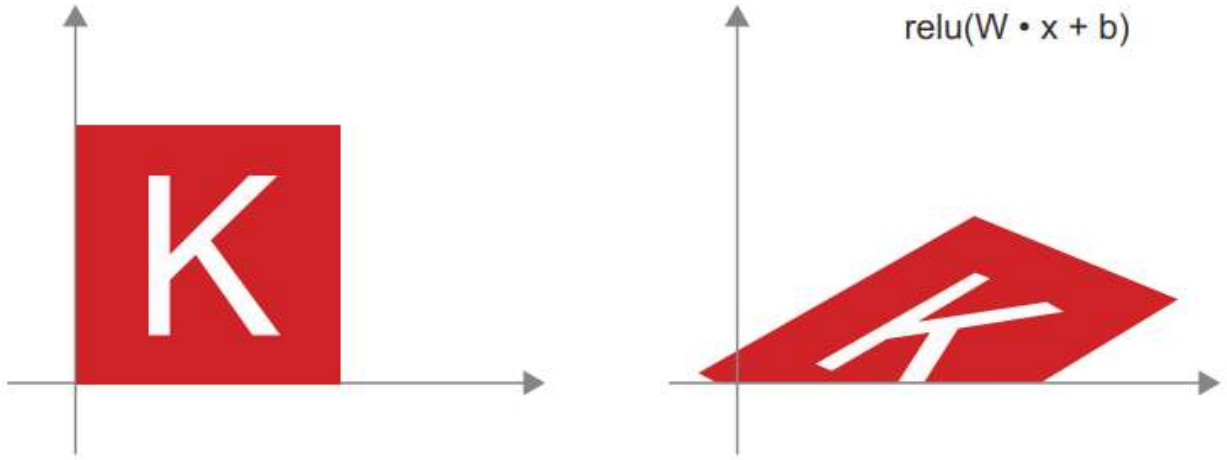
- That's an affine transform where the linear part is the matrix $\texttt{W2} \cdot \texttt{W1}$ and the translation part is the vector $\texttt{W2} \cdot \texttt{b1} + \texttt{b2}$.

`Dense` Layer with `relu` Activation

- As a consequence, a multilayer neural network made entirely of `Dense` layers without activations would be equivalent to a single `Dense` layer.
- This “deep” neural network would just be a linear model in disguise!

`Dense` Layer with `relu` Activation

- This is why we need activation functions, like `relu`.
- Thanks to activation functions, a chain of `Dense` layers can be made to implement very complex, non-linear geometric transformations, resulting in very rich hypothesis spaces for your deep neural networks.



Affine transform followed by relu activation

A Geometric Interpretation of Deep Learning

- Neural networks consist entirely of chains of tensor operations.
- That these tensor operations are just simple geometric transformations of the input data.
- It follows that you can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a series of simple steps.

A Geometric Interpretation of Deep Learning

- In 3D, the following mental image may prove useful:
  - Imagine two sheets of colored paper: one red and one blue.
  - Put one on top of the other.
  - Now crumple them together into a small ball.
  - That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem.

A Geometric Interpretation of Deep Learning

- What a neural network is meant to do is figure out a transformation of the paper ball that would uncrumple it, so as to make the two classes cleanly separable again.
- With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.

Uncrumpling a complicated manifold of data

A Geometric Interpretation of Deep Learning

- Uncrumpling paper balls is what deep learning is about:
  - Finding neat representations for complex, highly folded data manifolds in high-dimensional spaces (a manifold is a continuous surface, like our crumpled sheet of paper).

A Geometric Interpretation of Deep Learning

- Intuition to why deep learning excels at this:
  - it takes the approach of incrementally decomposing a complicated geometric transformation into a long chain of elementary ones, which is pretty much the strategy a human would follow to uncrumple a paper ball.
  - Each layer in a deep network applies a transformation that disentangles the data a little, and a deep stack of layers makes tractable an extremely complicated disentanglement process.

The Engine of Neural Networks: Gradient- based Optimization

Gradient- based Optimization

- Each neural layer from our model example transforms its input data as follows:

```
output = relu(dot(input, W) + b)
```

- In this expression, `W` and `b` are tensors that are attributes of the layer.
- They're called the weights or trainable parameters of the layer (the kernel and bias attributes, respectively).
- These weights contain the information learned by the model from exposure to training data.

Gradient- based Optimization

- Initially, these weight matrices are filled with small random values (a step called random initialization).
- The resulting representations are meaningless—but they're a starting point.
- What comes next is to gradually adjust these weights, based on a feedback signal.
- This gradual adjustment, also called training, is the learning that machine learning is all about.

Gradient- based Optimization

- This happens within what's called a training loop, which works as follows.
- Repeat these steps in a loop, until the loss seems sufficiently low:
  1. Draw a batch of training samples, `x`, and corresponding targets, `y_true`.
  2. Run the model on `x` (a step called the forward pass) to obtain predictions, `y_pred`.
  3. Compute the loss of the model on the batch, a measure of the mismatch between `y_pred` and `y_true`.
  4. Update all weights of the model in a way that slightly reduces the loss on this batch.

Gradient- based Optimization

- Given an individual weight coefficient in the model, how can you compute whether the coefficient should be increased or decreased, and by how much?

Gradient- based Optimization

- One naive solution would be to freeze all weights in the model except the one scalar coefficient being considered, and try different values for this coefficient. This would have to be repeated for all coefficients in the model.
- But such an approach would be horribly inefficient, because you'd need to compute two forward passes (which are expensive) for every individual coefficient (of which there are many, usually thousands and sometimes up to millions).
- Thankfully, there's a much better approach: gradient descent.

Gradient- based Optimization

- One naive solution would be to freeze all weights in the model except the one scalar coefficient being considered, and try different values for this coefficient. This would have to be repeated for all coefficients in the model.
- But such an approach would be horribly inefficient, because you'd need to compute two forward passes (which are expensive) for every individual coefficient (of which there are many, usually thousands and sometimes up to millions).
- Thankfully, there's a much better approach: gradient descent.

Gradient- based Optimization

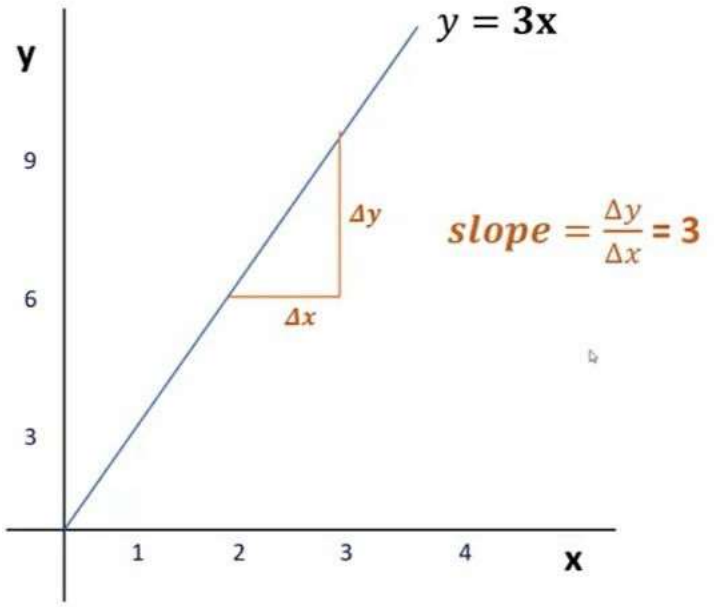
- Gradient descent is the optimization technique that powers modern neural networks.
- All of the functions used in our models (such as `dot` or `+`) transform their input in a smooth and continuous way:
  - if you look at `z = x + y`, for instance, a small change in `y` only results in a small change in `z`, and if you know the direction of the change in `y`, you can infer the direction of the change in `z`.
- Mathematically, you'd say these functions are differentiable.
- If you chain together such functions, the bigger function you obtain is still differentiable.

Gradient- based Optimization

- In particular, this applies to the function that maps the model's coefficients to the loss of the model on a batch of data:
  - a small change in the model's coefficients results in a small, predictable change in the loss value.
- This enables you to use a mathematical operator called the gradient to describe how the loss varies as you move the model's coefficients in different directions.
- If you compute this gradient, you can use it to move the coefficients (all at once in a single update, rather than one at a time) in a direction that decreases the loss.

The Slope

- In mathematics, slope refers to the steepness and direction of a line.
- It is calculated as the ratio of the vertical change (rise) to the horizontal change (run) between any two distinct points on the line.
- A positive slope means the line is increasing (going up from left to right),
- A negative slope means the line is decreasing (going down from left to right).

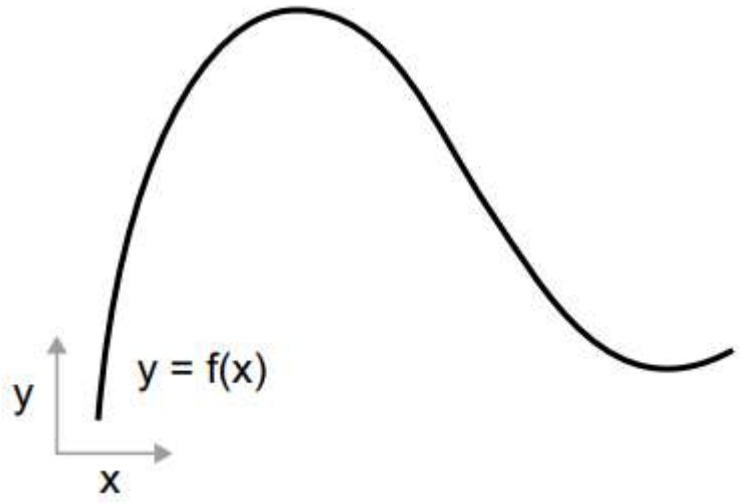


Slope vs Derivative vs Gradient

- In essence, slope, derivative, and gradient all relate to the concept of steepness or rate of change, but they apply to different contexts and mathematical objects.
- Slope specifically refers to the steepness of a straight line.
- Derivative is the rate of change of a function, which can be a curve.
- Gradient is the rate of change of a multivariable function, and it's a vector that points in the direction of the steepest ascent.

The Derivative

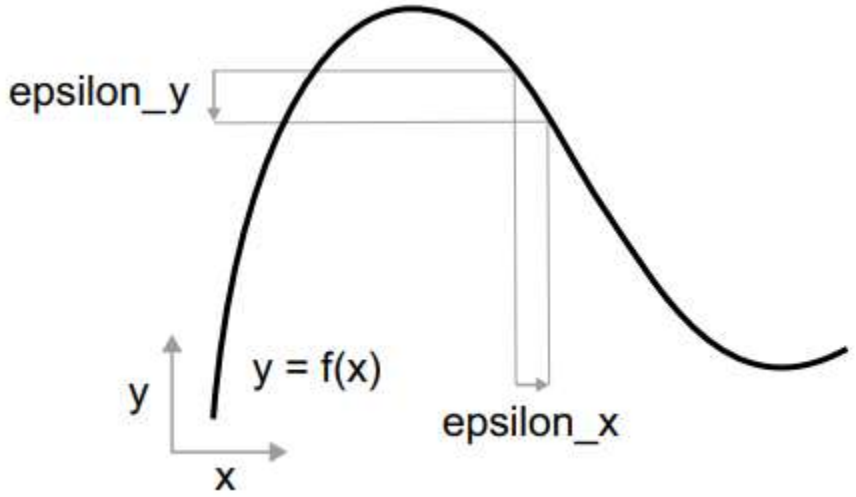
- Consider a continuous, smooth function `f(x) = y`, mapping a number, `x`, to a new number, `y`.



A continuous, smooth function

The Derivative

- Because the function is continuous, a small change in `x` can only result in a small change in `y`.
- Let's say you increase `x` by a small factor, `epsilon_x`: this results in a small `epsilon_y` change to `y`.

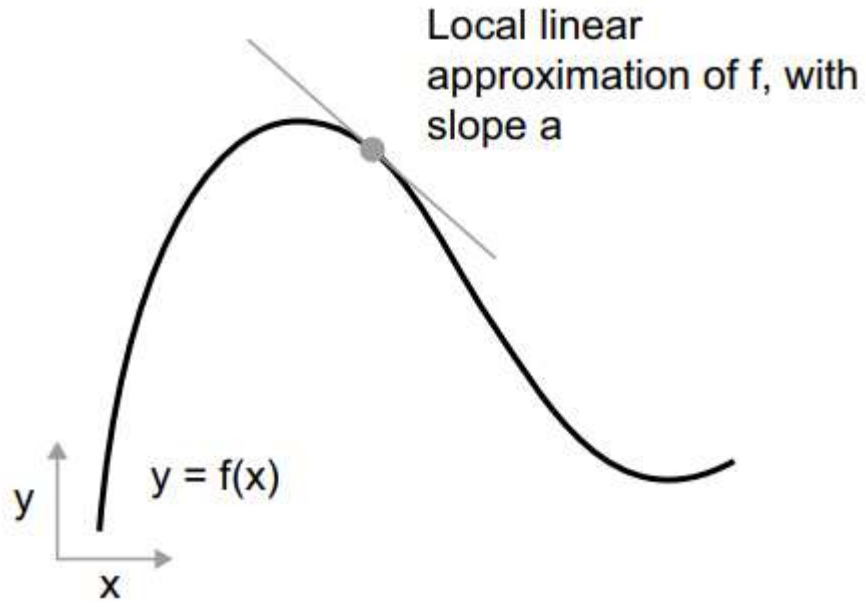


With a continuous function,

a small change in x results in a small change in y.

The Derivative

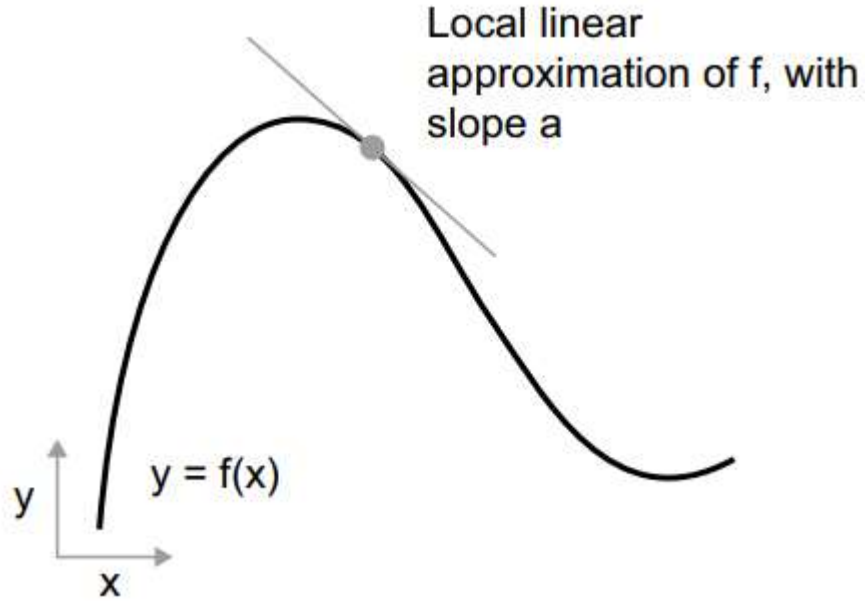
- Because the function is smooth (its curve doesn't have any abrupt angles), when `epsilon_x` is small enough, around a certain point `p`, it's possible to approximate `f` as a linear function of slope `a`, so that `epsilon_y` becomes `a * epsilon_x`:
  `f(x + epsilon_x) = y + a * epsilon_x`
- Obviously, this linear approximation is valid only when `x` is close enough to `p`.



Derivative of f in p

The Derivative

- The slope `a` is called the derivative of `f` in `p`.
- If `a` is negative, it means a small increase in `x` around `p` will result in a decrease of `f(x)`
- if `a` is positive, a small increase in `x` will result in an increase of `f(x)`.
- The absolute value of `a` (the magnitude of the derivative) tells you how quickly this increase or decrease will happen.



Derivative of f in p

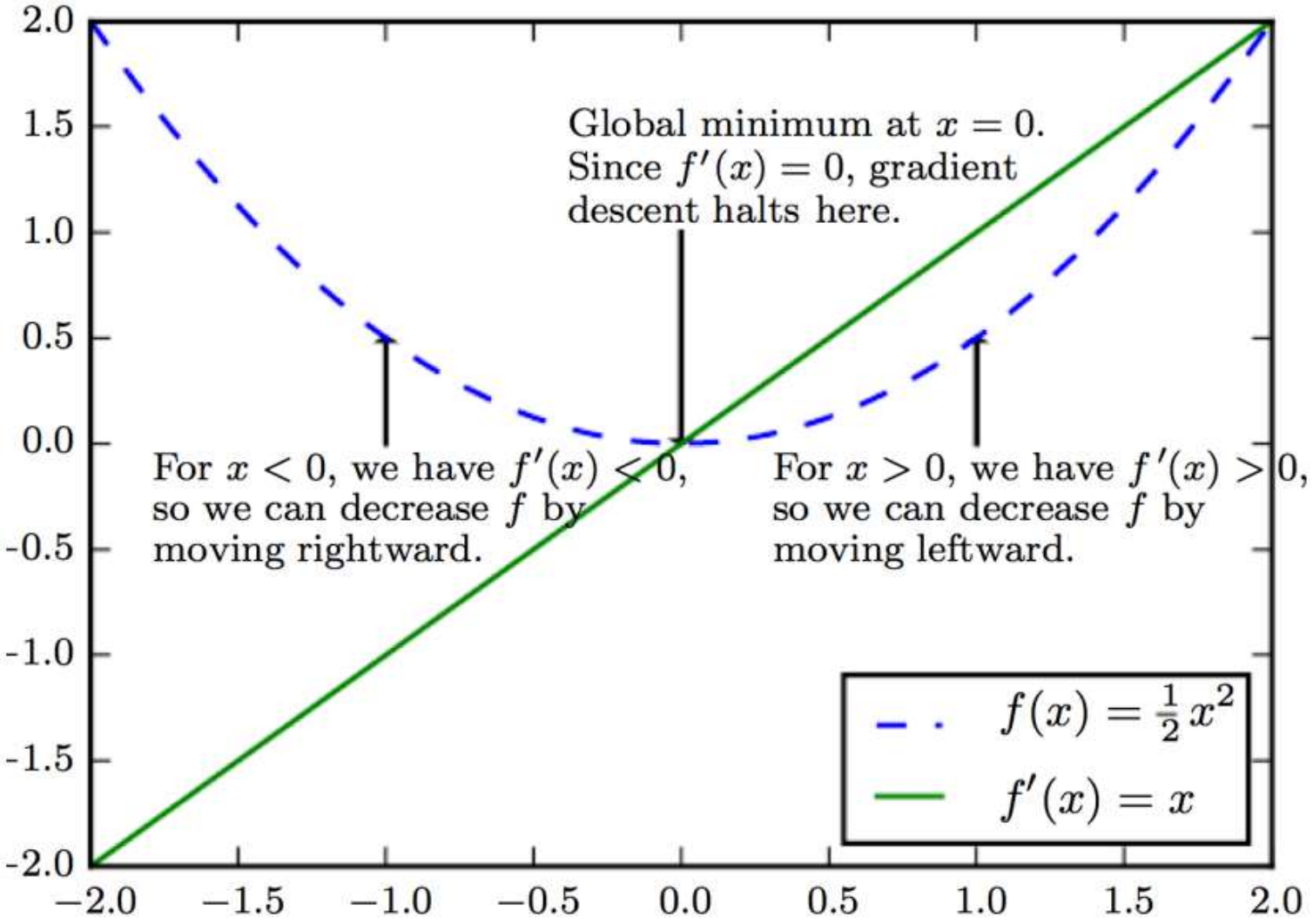
The Derivative

- For every differentiable function `f(x)` (differentiable means “can be derived”: for example, smooth, continuous functions can be derived), 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.
- For instance, the derivative of `cos(x)` is `-sin(x)`, the derivative of `f(x) = a * x` is `f'(x) = a`, and so on.

The Derivative

- Being able to derive functions is a very powerful tool when it comes to optimization, the task of finding values of `x` that minimize the value of `f(x)`.
- If you're trying to update `x` by a factor `epsilon_x` in order to minimize `f(x)`, and you know the derivative of `f`, then your job is done:
    - the derivative completely describes how `f(x)` evolves as you change `x`.
- If you want to reduce the value of `f(x)`, you just need to move `x` a little in the opposite direction from the derivative.

The Derivative



Use f’(x) to follow function downhill

Reduce f(x) by going in direction opposite sign of derivative f’(x)

The Gradient

- The previous function turned a scalar value `x` into another scalar value `y`:
  - you could plot it as a curve in a 2D plane.
- Now imagine a function that turns a tuple of scalars `(x, y)` into a scalar value `z`:
  - that would be a vector operation.
  - You could plot it as a 2D surface in a 3D space (indexed by coordinates `x, y, z`).
- Likewise, you can imagine functions that take matrices as inputs, functions that take rank-3 tensors as inputs, etc.

The Gradient

- The concept of derivation can be applied to any such function, as long as the surfaces they describe are continuous and smooth.
- The derivative of a tensor operation (or tensor function) is called a gradient.
- Gradients are just the generalization of the concept of derivatives to functions that take tensors as inputs.

The Gradient

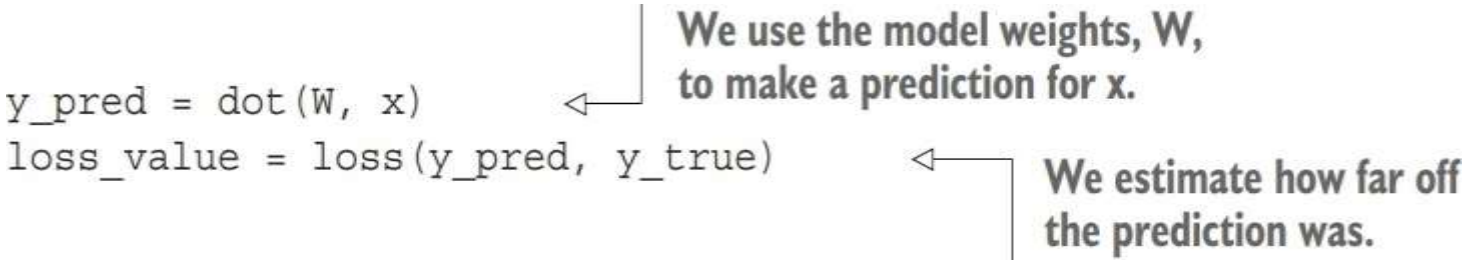
- For a scalar function, the derivative represents the local slope of the curve of the function.
- In the same way, the gradient of a tensor function represents the curvature of the multidimensional surface described by the function.
- It characterizes how the output of the function varies when its input parameters vary.

The Gradient

- Let's look at an example grounded in machine learning.
- Consider
  - An input vector, `x` (a sample in a dataset)
  - A matrix, `W` (the weights of a model)
  - A target, `y_true` (what the model should learn to associate to `x`)
  - A loss function, `loss` (meant to measure the gap between the model's current predictions and `y_true`)

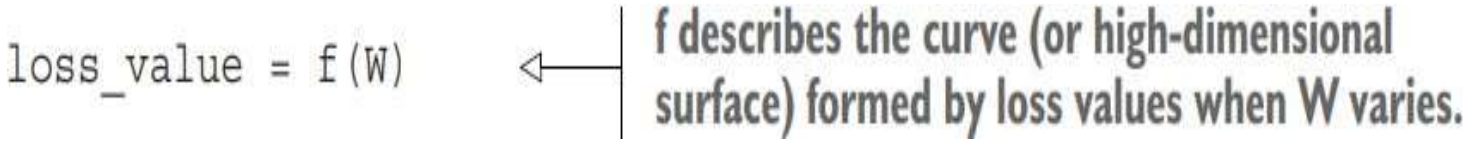
The Gradient

- You can use `W` to compute a target candidate `y_pred`, and then compute the loss, or mismatch, between the target candidate `y_pred` and the target `y_true`:



The Gradient

- Now we'd like to use gradients to figure out how to update `W` so as to make `loss_value` smaller. How do we do that?
- Given fixed inputs `x` and `y_true`, the preceding operations can be interpreted as a function mapping values of `W` (the model's weights) to loss values:



The Gradient

- Let's say the current value of `W` is `W0`.
- Then the derivative of `f` at the point `W0` is a tensor `grad(loss_value, W0)`, with the same shape as `W`, where each coefficient `grad(loss_value, W0)[i, j]` indicates the direction and magnitude of the change in `loss_value` you observe when modifying `W0[i, j]`.
- That tensor `grad(loss_value, W0)` is the gradient of the function `f(W) = loss_value` in `W0`, also called "gradient of `loss_value` with respect to `W` around `W0`."

The Gradient

- Concretely, what does `grad(loss_value, W0)` represent?
  - 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 `loss_value = f(W)` around `W0`, as well as the slope of this ascent.
  - Each partial derivative describes the slope of `f` in a specific direction.

The Gradient

- For a function `f(x)`, you can reduce the value of `f(x)` by moving `x` a little in the opposite direction from the derivative.
- Likewise, with a function `f(W)` of a tensor, you can reduce `loss_value = f(W)` by moving `W` in the opposite direction from the gradient:
  - for example, `W1 = W0 - step * grad(f(W0), W0)` (where step is a small scaling factor).
- That means going against the direction of steepest ascent of `f`, which intuitively should put you lower on the curve.
- Note that the scaling factor step is needed because `grad(loss_value, W0)` only approximates the curvature when you're close to `W0`, so you don't want to get too far from `W0`.

Stochastic Gradient Descent (SGD)

- Given a differentiable function, it's theoretically possible to find its minimum analytically:
  - it's known that a function's minimum is a point where the derivative is 0.
- So all you have to do is find all the points where the derivative goes to 0.
- Check for which of these points the function has the lowest value.

Stochastic Gradient Descent (SGD)

- Applied to a neural network, that means finding analytically the combination of weight values that yields the smallest possible loss function.
- This can be done by solving the equation grad(f(W), W) = 0 for W.
- This is a polynomial equation of N variables, where N is the number of coefficients in the model.
- Although it would be possible to solve such an equation for N = 2 or N = 3, doing so is intractable for real neural networks, where the number of parameters is never less than a few thousand and can often be several tens of millions.

Stochastic Gradient Descent (SGD)

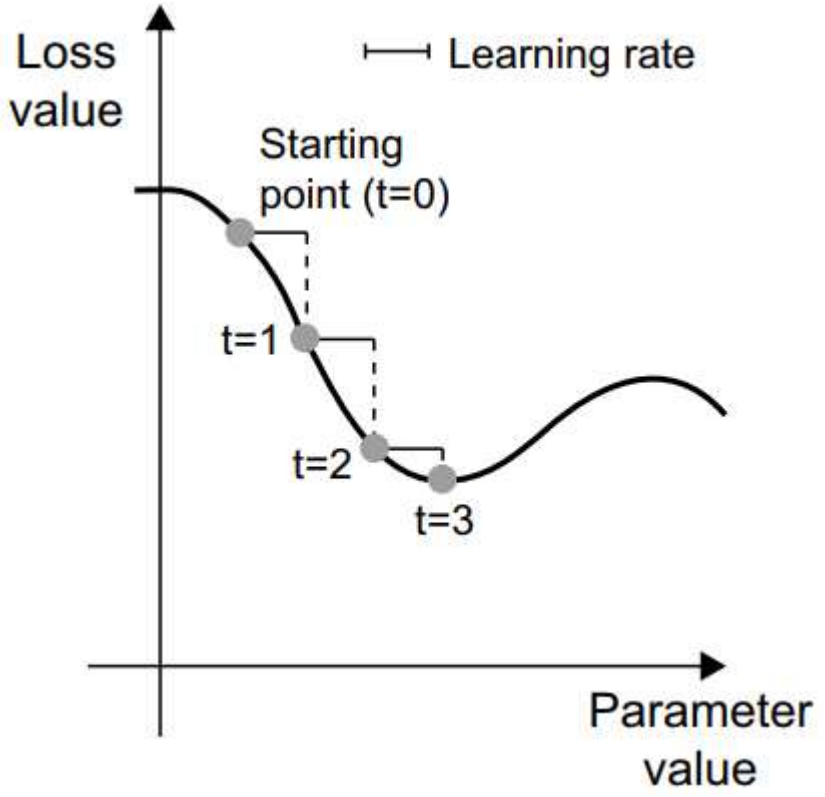
- Instead, you can use the four-step algorithm outlined at the beginning of this section:
  1. Draw a batch of training samples, x, and corresponding targets, y_true.
  2. Run the model on x to obtain predictions, y_pred (this is called the forward pass).
  3. Compute the loss of the model on the batch, a measure of the mismatch between y_pred and y_true.
  4. Compute the gradient of the loss with regard to the model's parameters (this is called the backward pass).
  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 (learning_rate here) would be a scalar factor modulating the "speed" of the gradient descent process.

Stochastic Gradient Descent (SGD)

- What we just described is called mini-batch stochastic gradient descent (mini-batch SGD).
- The term stochastic refers to the fact that each batch of data is drawn at random (stochastic is a scientific synonym of random).

Stochastic Gradient Descent (SGD)

- Figure 2.18 illustrates what happens in 1D, when the model has only one parameter and you have only one training sample.
- Intuitively it's important to pick a reasonable value for the learning_ rate factor.
- If it's too small, the descent down the curve will take many iterations, and it could get stuck in a local minimum.
- If learning_rate is too large, your updates may end up taking you to completely random locations on the curve.



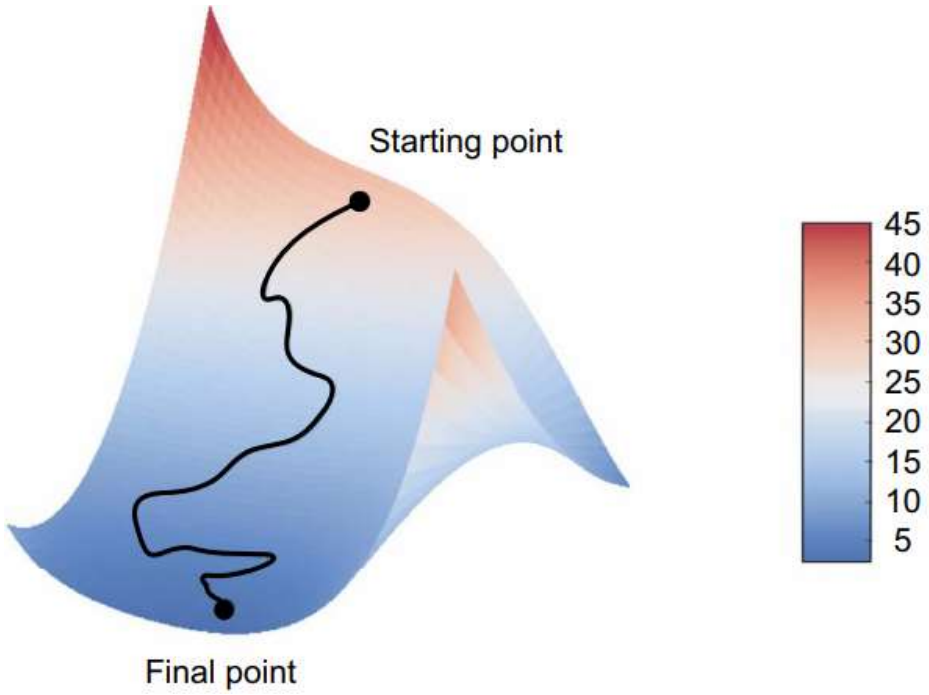
SGD down a 1D loss curve (one learnable parameter)

Stochastic Gradient Descent (SGD)

- Note that a variant of the mini-batch SGD algorithm would be to draw a single sample and target at each iteration, rather than drawing a batch of data. This would be true SGD (as opposed to mini-batch SGD).
- Alternatively, going to the opposite extreme, you could run every step on all data available, which is called batch gradient descent.
- Each update would then be more accurate, but far more expensive.
- The efficient compromise between these two extremes is to use mini-batches of reasonable size.

Stochastic Gradient Descent (SGD)

- Although figure 2.18 illustrates gradient descent in a 1D parameter space, in practice you'll use gradient descent in highly dimensional spaces: every weight coefficient in a neural network is a free dimension in the space, and there may be tens of thousands or even millions of them.
- To help you build intuition about loss surfaces, you can also visualize gradient descent along a 2D loss surface, as shown in figure.



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

Stochastic Gradient Descent (SGD)

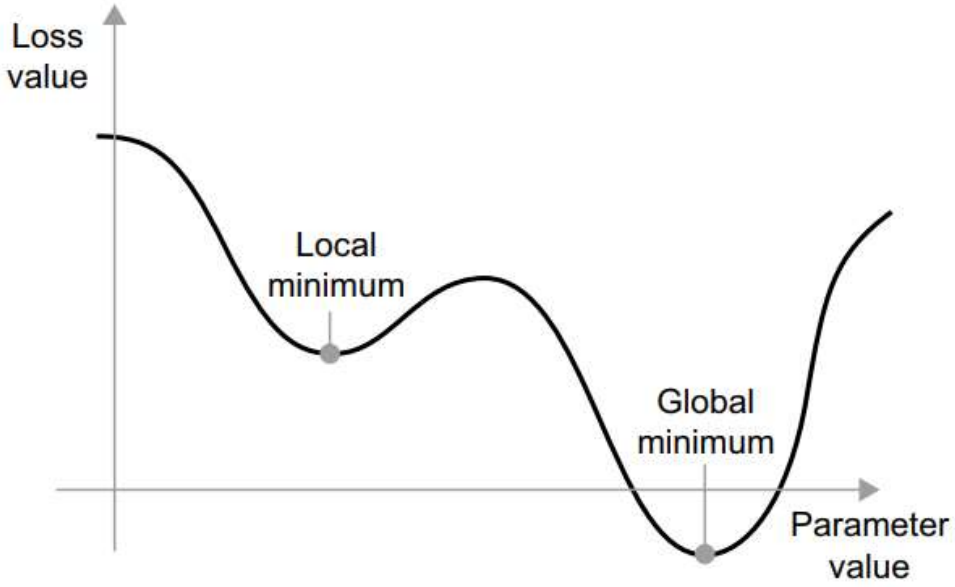
- But you can't possibly visualize what the actual process of training a neural network looks like—you can't represent a 1,000,000-dimensional space in a way that makes sense to humans.
- As such, it's good to keep in mind that the intuitions you develop through these low-dimensional representations may not always be accurate in practice.
- This has historically been a source of issues in the world of deep learning research.

Stochastic Gradient Descent (SGD)

- Additionally, 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.
- There is, for instance, SGD with momentum, as well as Adagrad, RMSprop, and several others.
- Such variants are known as optimization methods or optimizers.
- Many of these variants use the concept of momentum.
- Momentum addresses two issues with SGD: convergence speed and local minima.

Stochastic Gradient Descent (SGD)

- Consider figure 2.20, which shows the curve of a loss as a function of a model parameter.
- As you can see, around a certain parameter value, there is a local minimum: around that point, moving left would result in the loss increasing, but so would moving right.
- If the parameter under consideration were being optimized via SGD with a small learning rate, the optimization process could get stuck at the local minimum instead of making its way to the global minimum.



A local minimum and a global minimum

Stochastic Gradient Descent (SGD)

- You can avoid such issues by using momentum, which draws inspiration from physics.
- A useful mental image here is to think of the optimization process as a small ball rolling down the loss curve.
- If it has enough momentum, the ball won't get stuck in a ravine and will end up at the global minimum.
- Momentum is implemented by moving the ball at each step based not only on the current slope value (current acceleration) but also on the current velocity (resulting from past acceleration).
- In practice, this means updating the parameter w based not only on the current gradient value but also on the previous parameter update.

Chaining Derivatives:
The Backpropagation Algorithm

Chaining Derivatives:
The Backpropagation Algorithm

- Backpropagation 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.
- Crucially, a neural network consists of many tensor operations chained together, each of which has a simple, known derivative.

Chaining Derivatives:
The Backpropagation Algorithm

- For instance, A model defined can be expressed as a function parameterized by the variables W1, b1, W2, and b2 (belonging to the first and second Dense layers respectively), involving the atomic operations dot, relu, softmax, and +, as well as our loss function loss, which are all easily differentiable:
  loss_value =
  loss(y_true, softmax(dot(relu(dot(inputs, W1) + b1), W2) + b2))
- Calculus tells us that such a chain of functions can be derived using the following identity, called the chain rule.

Chaining Derivatives: The Backpropagation Algorithm

- The chain rule states that

```
grad(y, x) == grad(y, x1) * grad(x1, x).
```

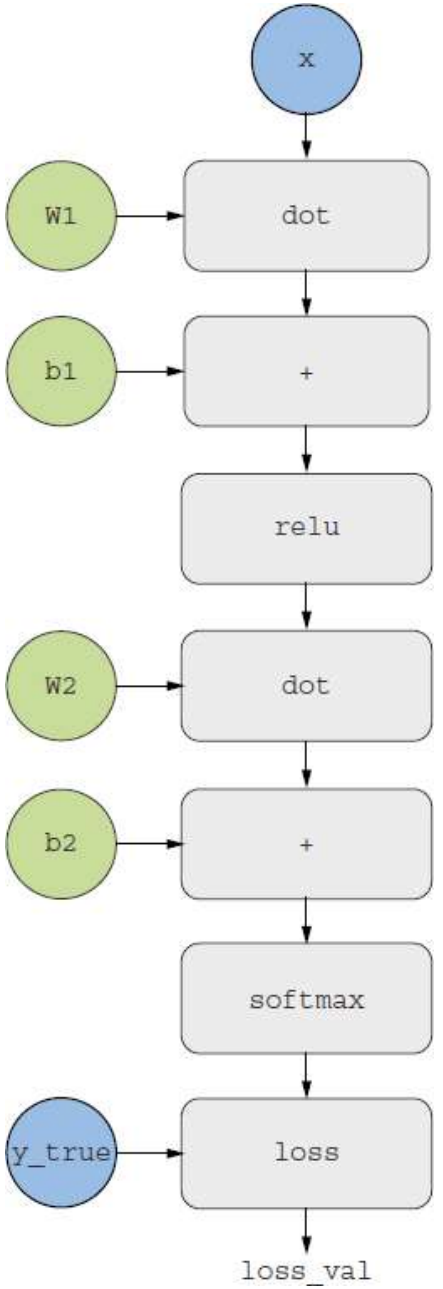
- The chain rule is named as it is because when you add more intermediate functions, it starts looking like a chain:

```
grad(y, x) ==
(grad(y, x3) * grad(x3, x2) * grad(x2, x1) * grad(x1, x))
```

- Applying the chain rule to the computation of the gradient values of a neural network gives rise to an algorithm called backpropagation.

Computation Graphs

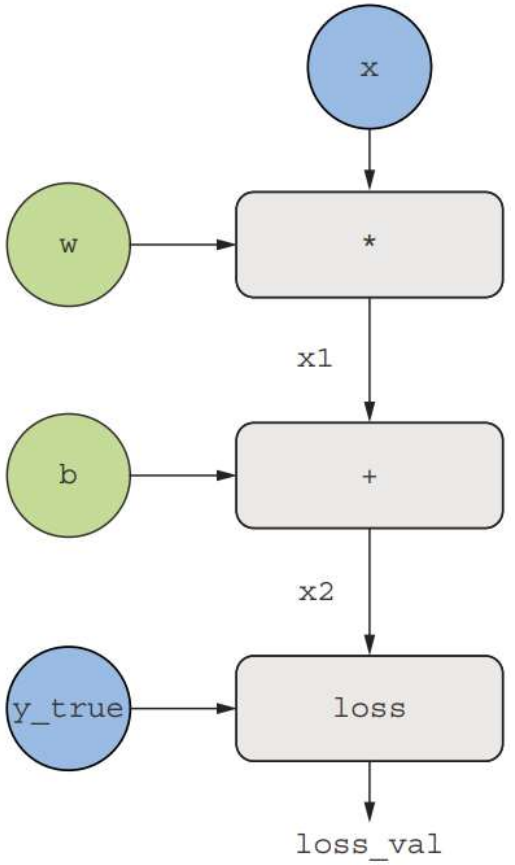
- A useful way to think about backpropagation is in terms of computation graphs.
- A Computation graph is a directed acyclic graph of operations—in our case, tensor operations.



The computation graph representation of our two-layer model

Computation Graphs

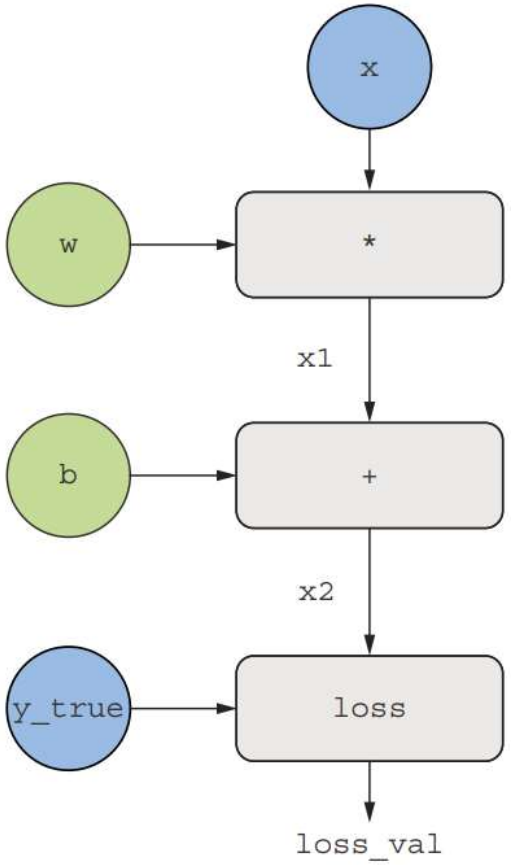
- Consider a simple computation graph, where we only have one linear layer and where all variables are scalar.
- We'll take two scalar variables w and b, a scalar input x, and apply some operations to them to combine them into an output y.



A basic example
of a computation graph

Computation Graphs

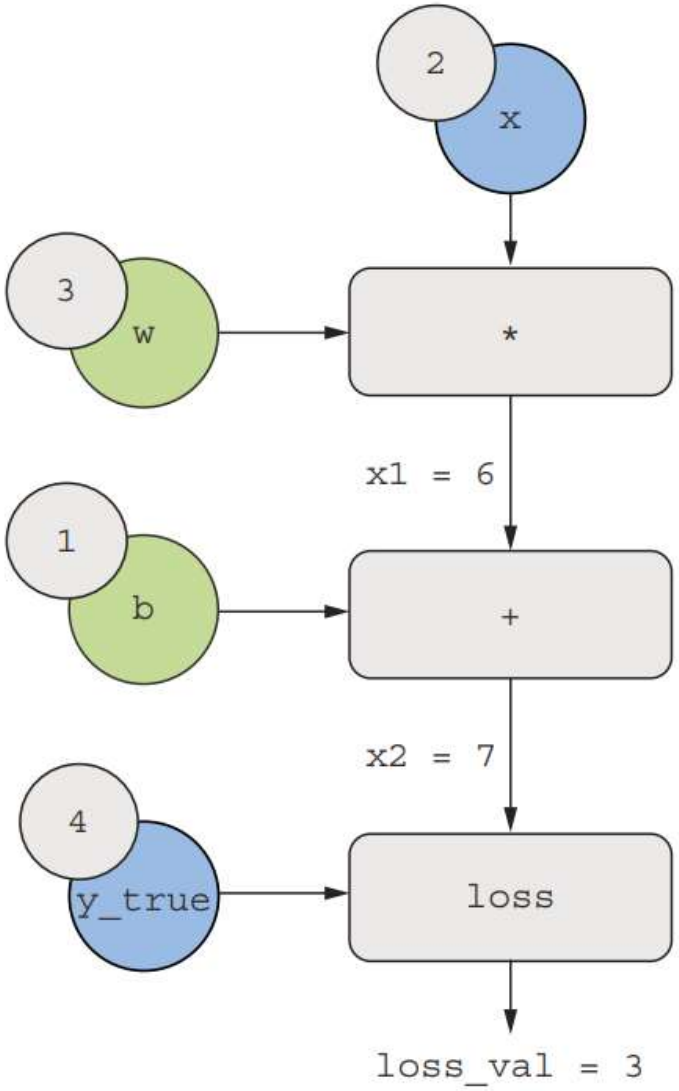
- Finally, we'll apply an absolute value error-loss function: loss_val = abs(y_true - y).
- Since we want to update w and b in a way that will minimize loss_val, we are interested in computing grad(loss_val, b) and grad(loss _val, w).



A basic example
of a computation graph

Forward Pass

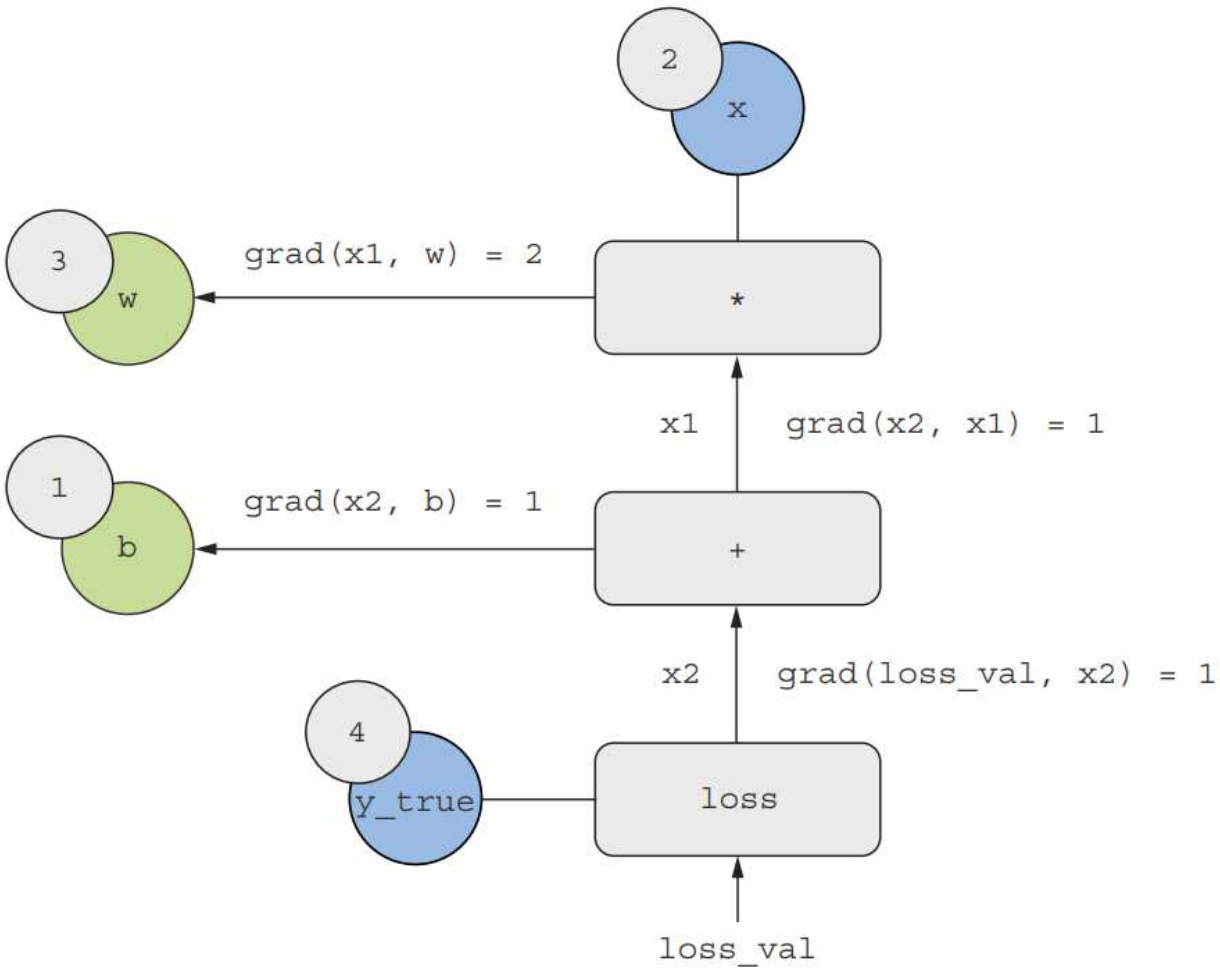
- Let's set concrete values for the "input nodes" in the graph, that is to say, the input x, the target y_true, w, and b.
- We'll propagate these values to all nodes in the graph, from top to bottom, until we reach loss_val.
- This is the forward pass.



Running a forward pass

Backward Pass

- Now let's "reverse" the graph: for each edge in the graph going from A to B, we will create an opposite edge from B to A, and ask, how much does B vary when A varies? That is to say, what is grad(B, A)?
- We'll annotate each inverted edge with this value.
- This backward graph represents the backward pass.



Running a backward pass

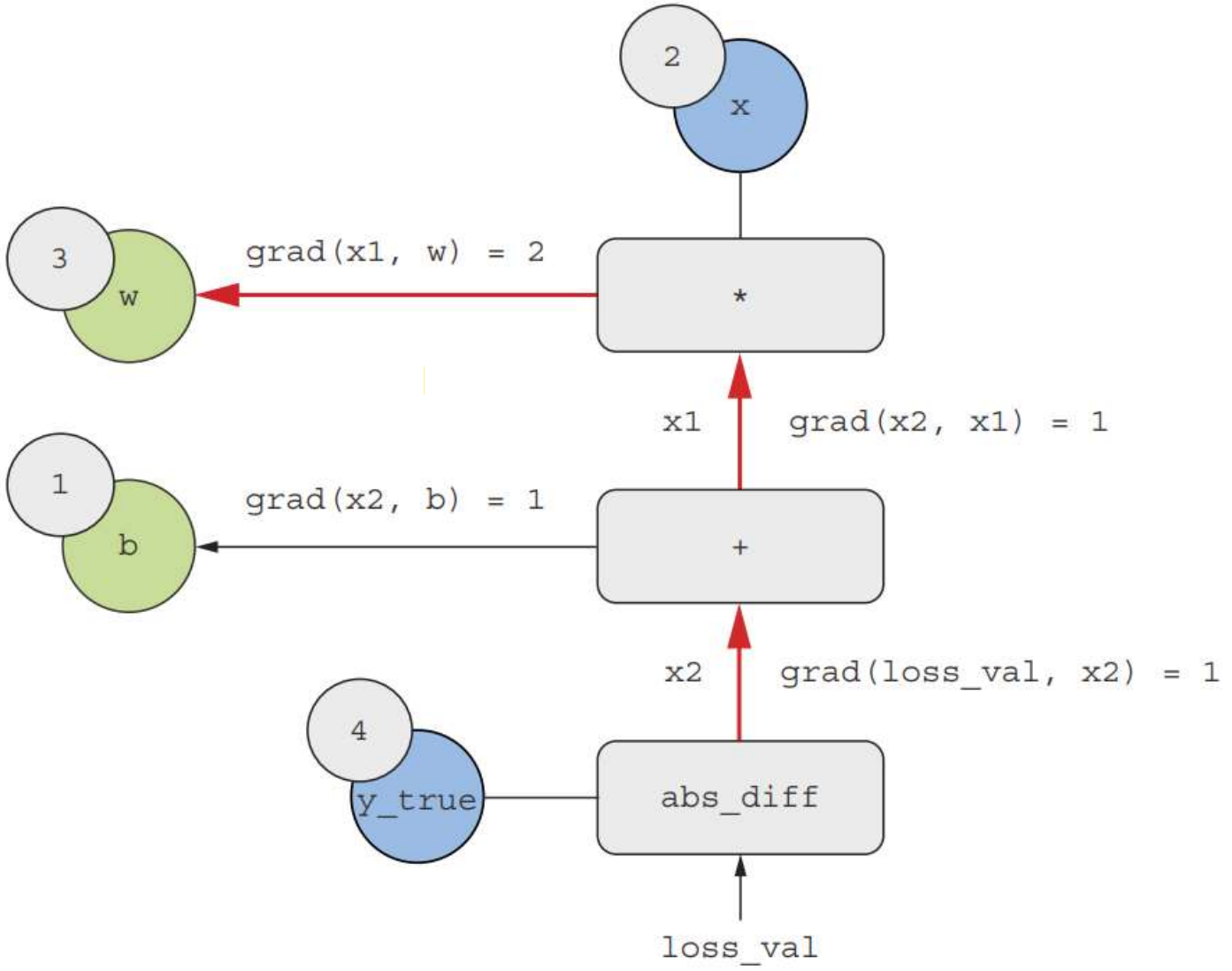
What is Obtained with Computation Graph?

We have the following:

- grad(loss_val, x2) = 1, because as x2 varies by an amount epsilon, loss_val = abs(4 - x2) varies by the same amount.
- grad(x2, x1) = 1, because as x1 varies by an amount epsilon, x2 = x1 + b = x1 + 1 varies by the same amount.
- grad(x2, b) = 1, because as b varies by an amount epsilon, x2 = x1 + b = 6 + b varies by the same amount.
- grad(x1, w) = 2, because as w varies by an amount epsilon, x1 = x * w = 2 * w varies by 2 * epsilon.

Path from loss_val to w in the backward graph

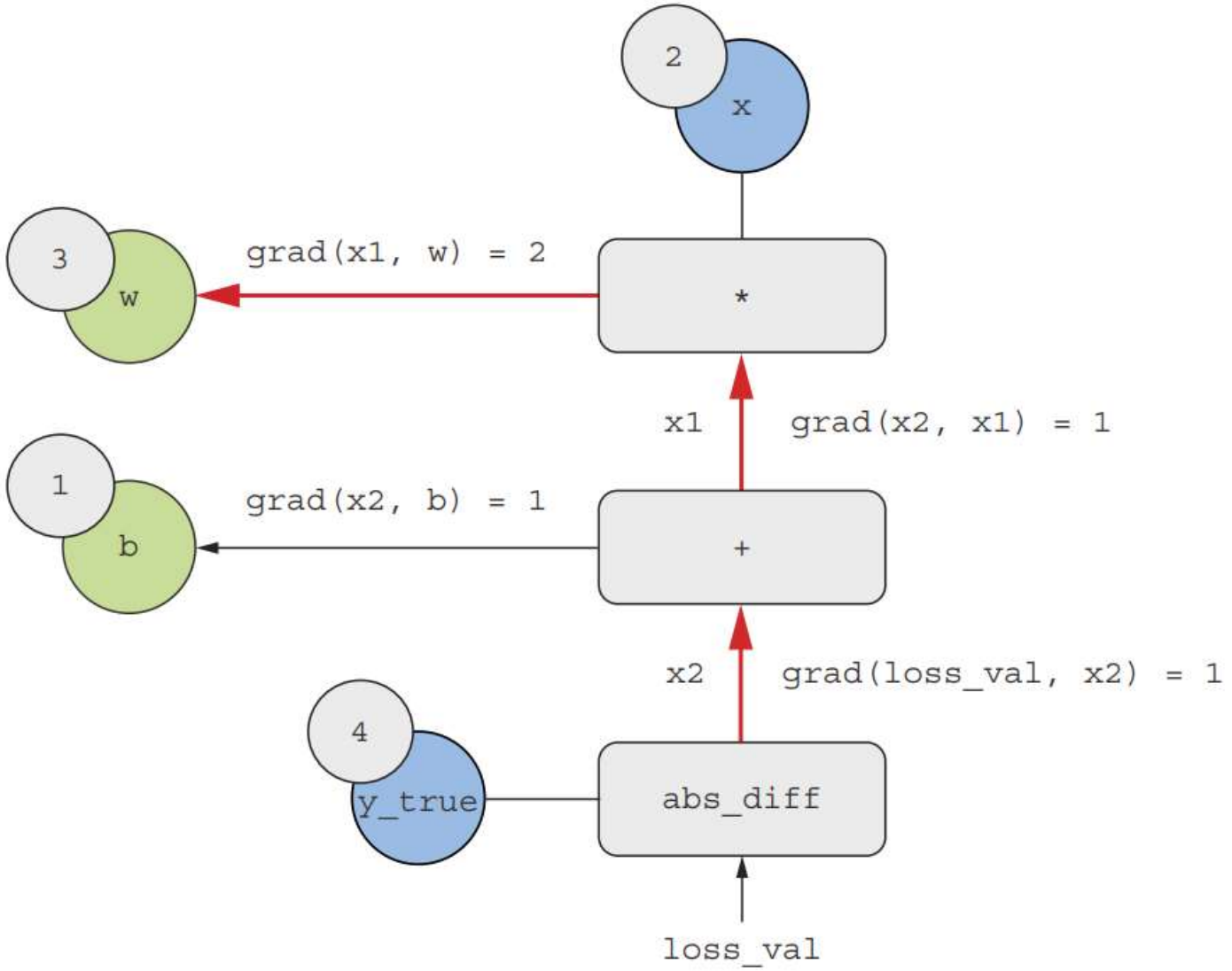
- What the chain rule says about this backward graph is that you can obtain the derivative of a node with respect to another node by multiplying the derivatives for each edge along the path linking the two nodes.



Path from `loss_val` to `w` in the backward graph

Path from loss_val to w in the backward graph

- For instance, grad(loss_val, w) = grad(loss_val, x2) * grad(x2, x1) * grad(x1, w)
- By applying the chain rule to our graph, we obtain what we were looking for:
  - grad(loss_val, w) = 1 * 1 * 2 = 2
  - grad(loss_val, b) = 1 * 1 = 1



Path from `loss_val` to `w` in the backward graph

Chaining Derivatives: The Backpropagation Algorithm

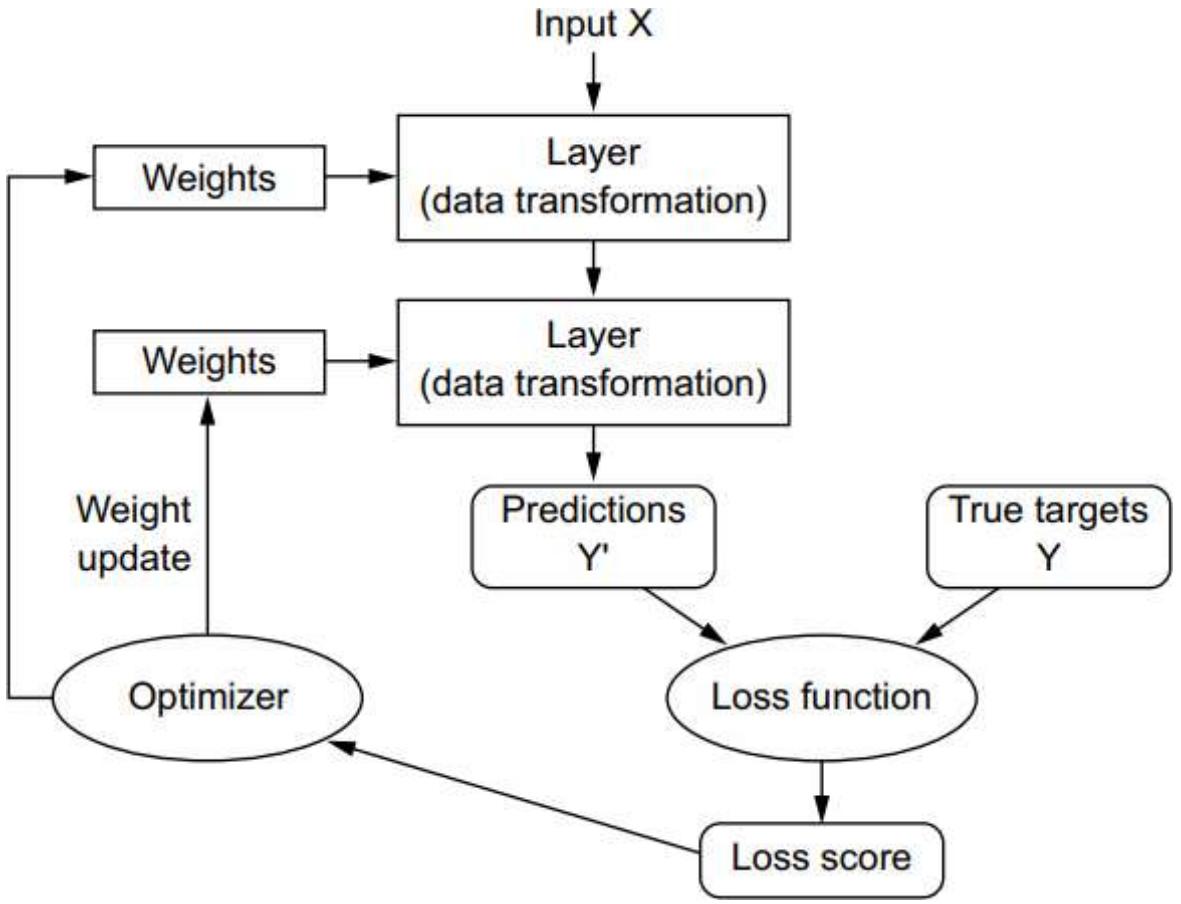
- And with that, you just saw backpropagation in action!
- Backpropagation is simply the application of the chain rule to a computation graph.
- There's nothing more to it.
- Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, computing the contribution that each parameter had in the loss value.
- That's where the name "backpropagation" comes from: we "back propagate" the loss contributions of different nodes in a computation graph.

Automatic Differentiation with Computation Graphs

- Modern frameworks, such as TensorFlow, are capable of automatic differentiation, which is implemented with the kind of computation graphs.

The network, layers, loss function, and optimizer

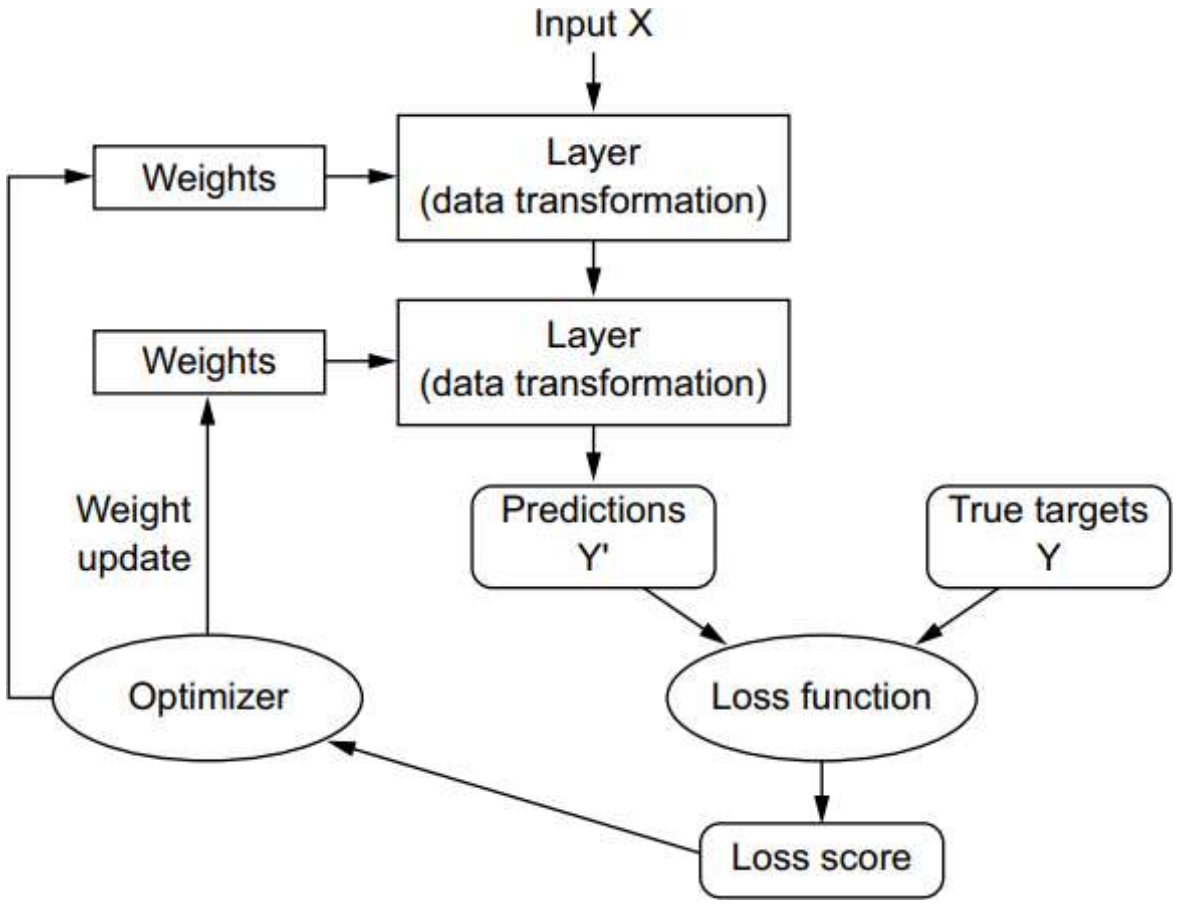
- The model, composed of layers that are chained together, maps the input data to predictions.



Relationship between the network, layers, loss function, and optimizer

The network, layers, loss function, and optimizer

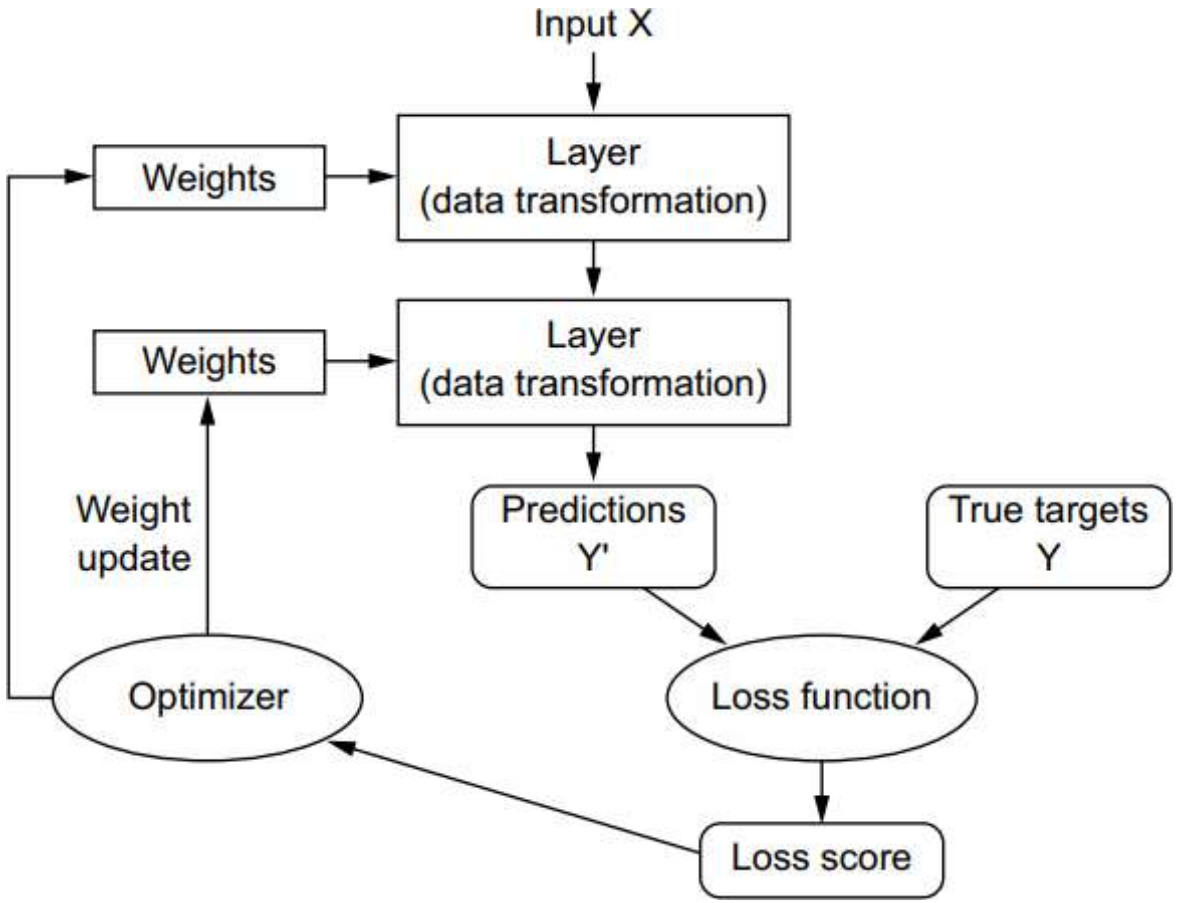
- The loss function then compares these predictions to the targets, producing a loss value: a measure of how well the model's predictions match what was expected.



Relationship between the network, layers, loss function, and optimizer

The network, layers, loss function, and optimizer

- The optimizer uses this loss value to update the model's weights.



Relationship between the network, layers, loss function, and optimizer

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).
- 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 set of values for the model’s weights that minimizes a loss function for a given set of training data samples and their corresponding targets.

Summary…

- Learning happens by drawing random batches of data samples and their targets, and computing the gradient of the model parameters with respect to the loss on the batch. The model 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 all tensor operations in neural networks are differentiable, and thus it's 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.

Summary…

- The two things (loss and optimizers) you need to define before you begin feeding data into a model.
  - The loss is 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.