1. First look at neural network

- Let's look at a concrete example of a neural network that uses the Python library Keras to learn to classify handwritten digits
  - ➢ We'll use the MNIST dataset, it's a set of 60,000 training images, plus 10,000 test images, assembled by the National Institute of Standards and Technology in the 1980s

- Load the images
  - ➢ The MNST dataset comes preloaded in Keras, in the form of a set of four NumPy arrays

Listing 2.1 Loading the MNIST dataset in Keras

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

  - ➢ The images are encoded as NumPy arrays, and the labels are an array of digits, ranging from 0 to 9, the images and labels have a one-to-one correspondence

Let's look at the training data:

```
>>> train_images.shape
(60000, 28, 28)
>>> len(train_labels)
60000
>>> train_labels
array([5, 0, 4, ..., 5, 6, 8], dtype=uint8)
```

And here's the test data:

```
>>> test_images.shape
(10000, 28, 28)
>>> len(test_labels)
10000
>>> test_labels
array([7, 2, 1, ..., 4, 5, 6], dtype=uint8)
```

- Build the network

Listing 2.2 The network architecture

```
from tensorflow import keras
from tensorflow.keras import layers
model = keras.Sequential([
    layers.Dense(512, activation="relu"),
    layers.Dense(10, activation="softmax")
])
```

- To make the model ready for training, we need to pick three more things as part of the compilation step
  - ➢ An optimizer – the mechanism through which the model will update itself based on the training data it sees, so as to improve its performance
  - ➢ A loss function – how the model will be able to measure its performance on the training data, and thus how it will be able to steer itself in the right direction
  - ➢ Metrics to monitor during training and testing – here, we'll only care about accuracy (the fraction of the images that were correctly classified)

Listing 2.3 The compilation step

```
model.compile(optimizer="rmsprop",
              loss="sparse_categorical_crossentropy",
              metrics=["accuracy"])
```

- We need to preprocess the data by reshaping it into the shape the model expects and scaling it so that all values are in the [0, 1] interval.
  - ➢ Previously, our training images were stored in an array of shape (60000, 28, 28) of type *unit8* with values in the [0, 255] interval. We'll transform it into a *float32* array of shape (60000, 28*28) with values between 0 and 1

Listing 2.4 Preparing the image data

```
train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

- We can train the model by calling the model's *fit()* method – we fit the model to its training data

Listing 2.5 "Fitting" the model

```
>>> model.fit(train_images, train_labels, epochs=5, batch_size=128)
Epoch 1/5
60000/60000 [===========================] - 5s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [=====================>.....] - ETA: 1s - loss: 0.1035 - acc: 0.9692
```

- Now we can use the trained model to predict class probabilities for new digits – images that weren't part of the training data

Listing 2.6 Using the model to make predictions

```
>>> test_digits = test_images[0:10]
>>> predictions = model.predict(test_digits)
>>> predictions[0]
array([1.0726176e-10, 1.6918376e-10, 6.1314843e-08, 8.4106023e-06,
       2.9967067e-11, 3.0331331e-09, 8.3651971e-14, 9.9999106e-01,
       2.6657624e-08, 3.8127661e-07], dtype=float32)
```

  - ➢ The first test digit has the highest probability score at index 7, so according to our model, it must be a 7

```
>>> predictions[0].argmax()
7
>>> predictions[0][7]
0.99999106
```

We can check that the test label agrees:

```
>>> test_labels[0]
7
```

- We can check the model's predictive power by assessing on average, how good is our model at classifying new digits?

Listing 2.7 Evaluating the model on new data

```
>>> test_loss, test_acc = model.evaluate(test_images, test_labels)
>>> print(f"test_acc: {test_acc}")
test_acc: 0.9785
```

  - ➢ The test-set accuracy is lower than the training-set accuracy. This gap is an example of overfitting.

2. Data representations for neural networks

- In the previous example, we started from data stored in multidimensional NumPy arrays, also called ***tensors***. In general, all current machine learning systems use tensors as their basic data structure
  - ➢ At its core, a tensor is a container for data – usually numerical data. So it's a container for numbers
  - ➢ Matrices are rank-2 tensors: tensors are a generalization of matrices to an arbitrary number of ***dimensions*** (note that in the context of tensors, a dimension is often called an ***axis***).

2.1. Scalars (rahnk-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* is a scalar tensor
  - ➢ You can display the number of axes of a NumPy tensor via the *ndim* attribute;
  - ➢ The number of axes of a tensor is also called its *rank*.

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

2.2. Vectors (rank-1 tensors)

- An array of numbers is called a *vector*, or rank-1 tensor, or 1D tensor.
    - ➢ A rank-1 tensor is said to have exactly one axis. Following is a NumPy vector

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

    - ➢ The vector has five entries and so is called a 5-dimensional vector. Don't confuse a 5D vector with a 5D tensor

2.3. Matrices (rank-2 tensors)

- An array of vector is a *matrix*, or rank-2 tensor, or 2D tensor.
    - ➢ The entries from the first axis are called the *rows*, and the entries from the second axis are called the *columns*.

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

2.4. Rank-3 and higher-rank tensors

- If matrices are packed into 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
```

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

2.5. Key attributes

- A tensor is defined by 3 key attributes

- Number of axes (ranks)
  - For instance, a rank-3 tensor has three axes, and a matrix has two axes. This is also called the tensor's *ndim* in Python libraries such as NumPy or TensorFlow
- Shape
  - This is a tuple of integers that describes how many dimensions the tensor has along each axis.
  - For instance, the previous matrix example has shape (3, 5), and the rank-3 tensor example has shape (3, 3, 5). A vector has a shape with a single element, such as (5,), whereas a scalar has an empty shape, ()
- Data type (usually called *dtype* in Python libraries)
  - This is the type of the data contained in the tensor; for instance, a tensor's type could be *float32, unit8*, …. In TensorFlow, you are also likely to come across string tensors

- Revisit the MNIST example
  - The dimension and data type

```
>>> train_images.ndim
3
```

Here's its shape:

```
>>> train_images.shape
(60000, 28, 28)
```

And this is its data type, the `dtype` attribute:

```
>>> train_images.dtype
uint8
```

  - So what we have here is a rank-3 tensor of 8-bit integers. More precisely, it's an array of 60,000 matrices of 28 * 28 integers. Each such matrix is a grayscale image, with coefficients between 0 and 255.
  - We can display the fourth digit in this rank-3 tensor, using the Matplotlib library

Listing 2.8 Displaying the fourth digit

```
import matplotlib.pyplot as plt
digit = train_images[4]
plt.imshow(digit, cmap=plt.cm.binary)
plt.show()
```

Naturally, the corresponding label is the integer 9:

```
>>> train_labels[4]
9
```

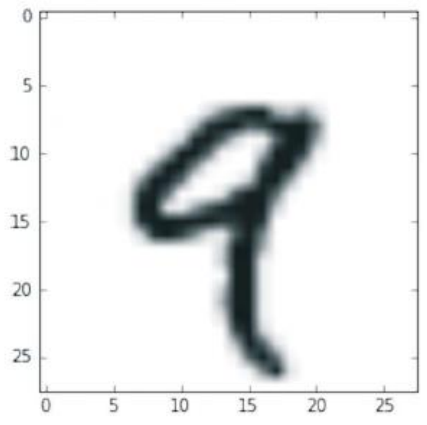


Figure 2.2 The fourth sample in our dataset

2.6. Manipulating tensors in NumPy

- In the previous example, we selected a specific digit alongside the first axis using the syntax *train_images[i]*.
  - Selecting specific elements in a tensor is called ***tensor slicing***.

- Let's look at the tensor-slicing operations you can do in NumPy arrays
  - The following example selects digits #10 to #100 (#100 isn't included) and puts them in an array of shape (90, 28, 28):

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

  - It's equivalent to this more detailed notation, which specifies a start index and stop index for the slice along each tensor axis. Note that : is equivalent to selecting the entire axis

```
>>> my_slice = train_images[10:100, :, :]        <- Equivalent to the previous example
>>> my_slice.shape
(90, 28, 28)
>>> my_slice = train_images[10:100, 0:28, 0:28]  <- Also equivalent to the previous example
>>> my_slice.shape
(90, 28, 28)
```

- In general, you may select slices between any two indices along each tensor axis.
  - For instance, in order to select 14*14 pixels in the bottom-right corner of all images, you could do this

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

  - And it is possible to use negative indices. Much like negative indices in Python lists, they indicate a position relative to the end of the current axis. In order to crop the images to patches of 14*14 pixels centered in the middle, you can do this

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

2.7. The notion of data batches

- In general, the first axis (axis 0) in all data tensors you'll come across in deep learning will be the *sample* axis. In the MNIST example, "samples" are images of digits.
- In addition, deep learning models don't process an entire dataset at once; rather, they break the data into small batches.
  - ➢ Concretely, here's one batch of our MNST digits, with a batch size of 128

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

    And here's the next batch:

    ```
    batch = train_images[128:256]
    ```

    And the *n*th batch:

    ```
    n = 3
    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.

2.8. Real-world examples of data tensors

- The data you'll manipulate will almost always fail into one of the following categories
  - ➢ **Vector** data – Rank-2 tensors of shape $(samples,\ features)$, where each sample is a vector of numerical attributes ("features")
    - ✧ Each single data point can be encoded as a vector, and thus a batch of data will be encoded as a rank-2 tensor, where the first axis is the *samples axis* and the second axis is the *features axis*.
    - ✧ E.g. 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, and thus an entire dataset of 100,000 people can be stored in a rank-2 tensor of shape (10000, 3)
  - ➢ **Timeseries** data or sequence data – Rank-3 tensors of shape $(samples,\ timesteps,\ features)$, where each sample is a sequence (of length *timestep*) of feature vectors
    - ✧ 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

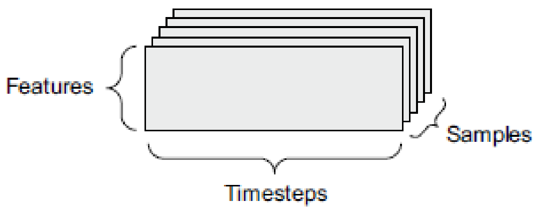


Figure 2.3 A rank-3 timeseries data tensor

    - ✧ The time axis is always the second axis (axis of index 1) by convention.

        - ✧ E.g. 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

- ➢ **Images** – Rank-4 tensors of shape $(samples, height, width, channels)$, where each sample is a 2D grid of pixels, and each pixel is represented by a vector of values ("channels")
    - ✧ Images typically have three dimensions: height, width, and color depth. A batch of 128 colored images of size 256*256 could thus be stored in a tensor of shape (128, 256, 256, 3)

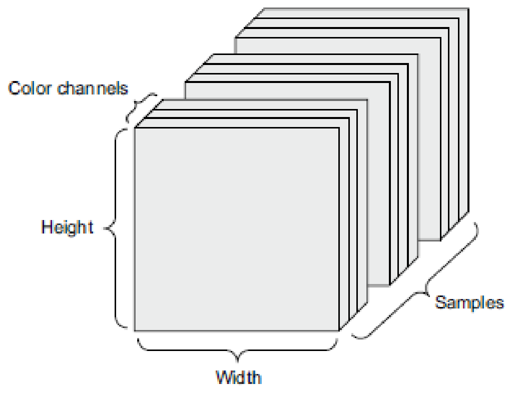


Figure 2.4 A rank-4 image data tensor

    - ✧ There are two conventions for shapes of image tensors
        - ▫ The *channels-last* convention (started with TensorFlow) places the color-depth axis at the end: $(samples, height, width, color\_depth)$
        - ▫ The *channels-first* convention (increasingly falling out of favor) places the color depth axis right after the batch axis: $(samples, color\_depth, height, width)$

- ➢ Video – Rank-5 tensors of shape $(samples, frames, height, width, channels)$, where each sample is a sequence (of length *frames*) of images
    - ✧ 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, a sequence of frames can be stored in a rank-4 tensor, and thus a batch of different videos can be stored in a rank-5 tensor.
    - ✧ E.g. 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 that 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)

3. *The gears: Tensor operation*

- Much like many computer program can be ultimately reduced to a small set of binary operations on binary inputs (e.g. AND, OR), all transformation learned by deep neural networks can be reduced to a handful of ***tensor operations*** applied to tensors of numeric data.

- In our initial example, we built our model by stacking dense layers on top of each other. A Keras layer instance looks like this

```
keras.layers.Dense(512, activation="relu")
```

  - ➢ This layer can be interpreted as a function, which takes an input a matrix and returns another matrix – a new representation for the input tensor.
  - ➢ Specifically, the function is as follows (*W* is a matrix and *b* is a vector, both attributes of the layer)

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

3.1. 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 (*vectorized* implementations).
  - ➢ Some python code to implement relu operation and addition operation

```
def naive_relu(x):
    assert len(x.shape) == 2

    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```

x is a rank-2 NumPy tensor.

Avoid overwriting the input tensor.

You could do the same for addition:

```
def naive_add(x, y):
    assert len(x.shape) == 2
    assert x.shape == y.shape
    x = x.copy()
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] += y[i, j]
    return x
```

x and y are rank-2 NumPy tensors.

Avoid overwriting the input tensor.

- In practice, when dealing with NumPy arrays, these operations are available as well-optimized built-in NumPy functions, which themselves delegate the heavy lifting to a Basic Linear Algebra Subprograms (BLAS) implementation.
  - ➢ BLAS are low-level, highly parallel, efficient tensor-manipulation routines that are typically implemented in Fortran or C
- Thus, in NumPy, you can do the following element-wise operation, and it will be blazing fast

```
import numpy as np
z = x + y
z = np.maximum(z, 0.)
```

Element-wise addition

Element-wise relu

Let's actually time the difference:

```
import time

x = np.random.random((20, 100))
y = np.random.random((20, 100))

t0 = time.time()
for _ in range(1000):
    z = x + y
    z = np.maximum(z, 0.)
print("Took: {0:.2f} s".format(time.time() - t0))
```

This takes 0.02 s. Meanwhile, the naive version takes a stunning 2.45 s:

```
t0 = time.time()
for _ in range(1000):
    z = naive_add(x, y)
    z = naive_relu(z)
print("Took: {0:.2f} s".format(time.time() - t0))
```

- Likewise, when running TensorFlow code on a GPU, element-wise operations are executed via fully vectorized CUDA implementations that can best utilize the highly parallel GPU chip architecture

3.2. Broadcasting

- Our earlier naïve implementation of addition only supports the addition of rank-2 tensors with identical shapes. But in the dense layer introduced earlier, we added a rank-2 tensor with a vector. What happens with addition when the shapes of the two tensors being added differ?
  - ➢ When possible, and without ambiguity, the smaller tensor will be *broadcast* to match the shape of the larger tensor. It consists of two steps
    - ✧ 1. Axes (called *broadcast axes*) are added to the smaller tensor to match the *ndim* of the larger tensor
    - ✧ 2. The smaller tensor is repeated alongside these new axes to match the full shape of the larger tensor
  - ➢ A concrete example with X shape (32, 10) and y shape (10,)

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

X is a random matrix with shape (32, 10).

y is a random vector with shape (10,).

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

```
y = np.expand_dims(y, axis=0)
```

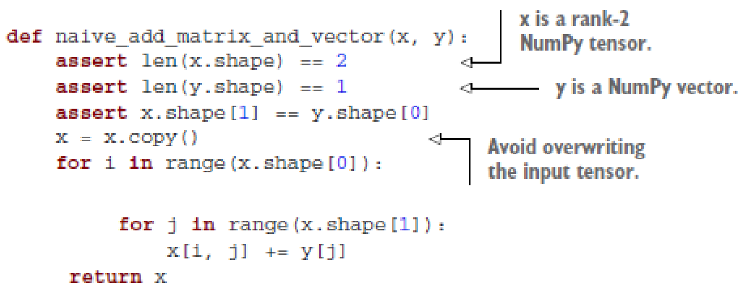
The shape of y is now (1, 10).

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)
```

Repeat y 32 times along axis 0 to obtain Y, which has shape (32, 10).

- In terms of implementation, no new rank-2 tensor is created, because that would be terribly inefficient. The repetition operation is entirely virtual: it happens at the algorithmic level rather than at the memory level. Here's what a naïve implementation look like

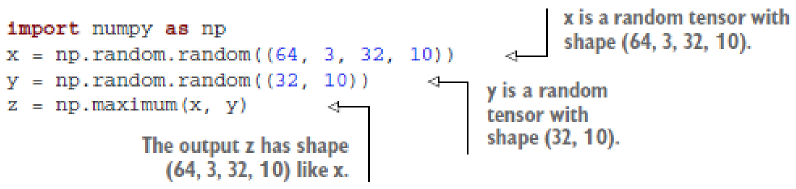


```python
def naive_add_matrix_and_vector(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    x = x.copy()
    for i in range(x.shape[0]):

        for j in range(x.shape[1]):
            x[i, j] += y[j]
    return x
```

x is a rank-2 NumPy tensor.
y is a NumPy vector.
Avoid overwriting the input tensor.

- 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*.
    - ➢ The following example applies the element-wise maximum operation to two tensors of different shape via broadcasting

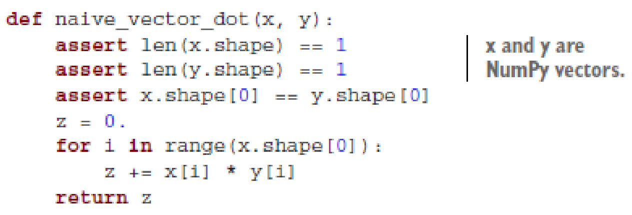


```python
import numpy as np
x = np.random.random((64, 3, 32, 10))
y = np.random.random((32, 10))
z = np.maximum(x, y)
```

x is a random tensor with shape (64, 3, 32, 10).
y is a random tensor with shape (32, 10).
The output z has shape (64, 3, 32, 10) like x.

3.3. Tensor product

- Mathematically, what does the dot operation do? The following piece of code demonstrates the logic behind tensor product



```python
def naive_vector_dot(x, y):
    assert len(x.shape) == 1
    assert len(y.shape) == 1
    assert x.shape[0] == y.shape[0]
    z = 0.
    for i in range(x.shape[0]):
        z += x[i] * y[i]
    return z
```

    - ➢ Notice that the dot product between two vectors is a scalar and that only vectors with the same number of elements are compatible for a dot product.
- One can also take 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.
    - ➢ It is implemented as follows

```
def naive_matrix_vector_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 1
    assert x.shape[1] == y.shape[0]
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            z[i] += x[i, j] * y[j]
    return z
```

x is a NumPy matrix.

y is a NumPy vector.

The first dimension of x must be the same as the 0th dimension of y!

This operation returns a vector of 0s with the same shape as y.

➢ Alternatively, we could use the function defined before

```
def naive_matrix_vector_dot(x, y):
    z = np.zeros(x.shape[0])
    for i in range(x.shape[0]):
        z[i] = naive_vector_dot(x[i, :], y)
    return z
```

➢ Note that as soon as one of the two tensors has a *ndim* greater than 1, *dot* is no longer symmetric, which is to say that $dot(x,\ y) \neq dot(y,\ x)$

- 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 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
    ➢ Here is a naïve implementation

```
def naive_matrix_dot(x, y):
    assert len(x.shape) == 2
    assert len(y.shape) == 2
    assert x.shape[1] == y.shape[0]
    z = np.zeros((x.shape[0], y.shape[1]))
    for i in range(x.shape[0]):
        for j in range(y.shape[1]):
            row_x = x[i, :]
            column_y = y[:, j]
            z[i, j] = naive_vector_dot(row_x, column_y)
    return z
```

x and y are NumPy matrices.

The first dimension of x must be the same as the 0th dimension of y!

Iterates over the rows of x . . .

. . . and over the columns of y.

This operation returns a matrix of 0s with a specific shape.

- To better understand dot-produce shape compatibility, consider

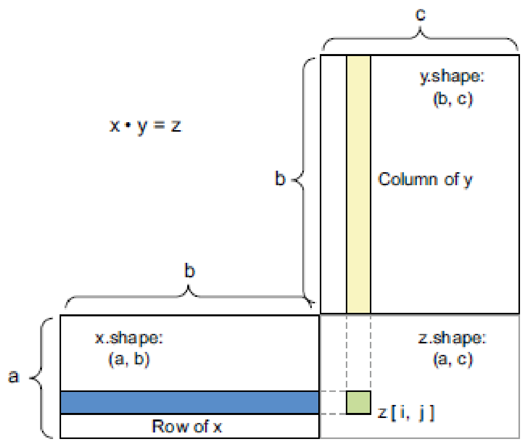


Figure 2.5 Matrix dot-product box diagram

- 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)
```

3.4. Tensor reshaping

- Although it wasn't used in the dense layers in our first neural network, we used it when we preprocessed the digits data before feeding it into our model

```
train_images = train_images.reshape((60000, 28 * 28))
```

- 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.
  - ➢ Consider the following examples

```
>>> 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.]])
```

3.5. Geometric interpretation of tensor operations

- Because the contents of the tensors manipulated by tensor operations can be interpreted as coordinates of points in some geometric space, all tensor operations have a geometric interpretation.
  - ➢ Consider **addition** of A [0.5, 1] and B [1, 0.25]

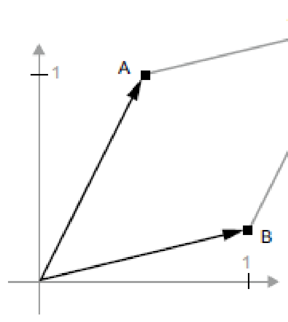


Figure 2.8 Geometric interpretation of the sum of two vectors

- ✧ This is done geometrically by representing the sum of the previous two vectors.
- ✧ 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.

- 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

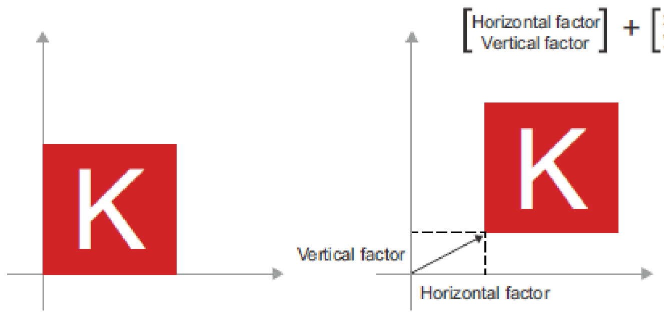


Figure 2.9 2D translation as a vector addition

- We can also perform
  - ➢ ***Rotation***: a counterclockwise rotation of a 2D vector by an angle theta can be achieved via a dot product with a 2*2 matrix $R = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$

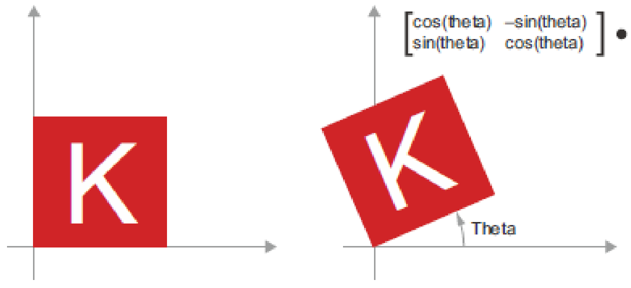


Figure 2.10 2D rotation (counterclockwise) as a dot product

  - ➢ ***Scaling***: a vertical and horizontal scaling of the image can be achieved via a dot product with a 2*2 matrix $S = \begin{bmatrix} horizontal\_factor & 0 \\ 0 & vertical\_factor \end{bmatrix}$. This is an example of diagonal matrix

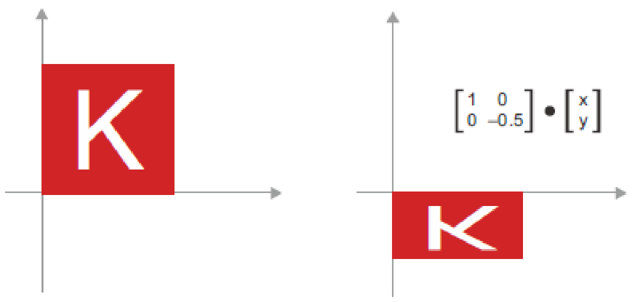

$\begin{bmatrix}1 & 0\\0 & -0.5\end{bmatrix} \bullet \begin{bmatrix}x\\y\end{bmatrix}$

Figure 2.11 2D scaling as a dot product

- ➢ ***Linear transformation***: a dot product with an arbitrary matrix implements a linear transform. Note that *scaling* and *rotation*, are by definition linear transforms
- ➢ ***Affine transformation***: an affine transform is the combination of a linear transform (achieved via a dot product with some matrix) and a translation (achieved via a vector addition). This is exactly the $y = W * x + b$ computation implemented by the dense layer
  - ✧ A dense layer without an activation function is an affine layer

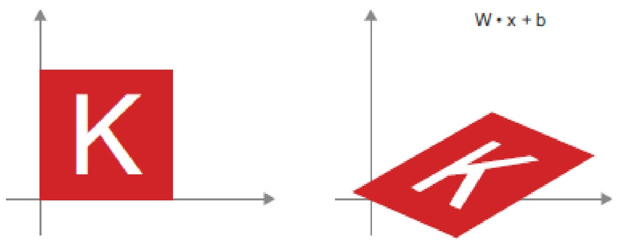


Figure 2.12 Affine transform in the plane

- Note that 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!
  - ➢ This is why we need activation functions , like ***relu***.
    - ✧ So that 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 network.

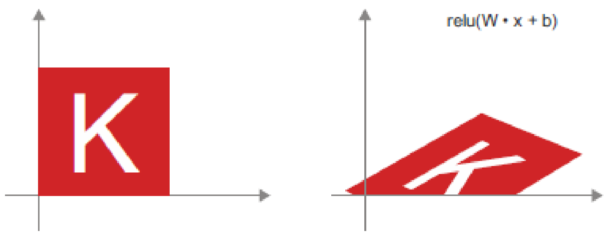


Figure 2.13 Affine transform followed by `relu` activation

3.6. Geometric Interpretation of deep learning

- Neural networks basically consist entirely of chains of tensor operations, and 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

- In 3D, the following illustration may be useful.

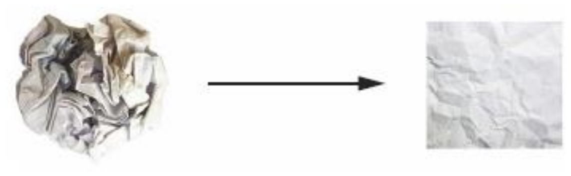


Figure 2.14 Uncrumpling a complicated manifold of data

  - Two sheets of colored paper: one red and one blue, put on top of the other, which crumpled 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.
  - 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 our fingers, one movement at a time.
- Uncrumpling paper balls is what machine 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).
  - That's 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.

4. *The Engine: Gradient-based optimization*

- In our expression $output = relu(dot(input,\ W) + b)$, 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.
  - Initially, these weights matrices are filled with small random values (a step called *random initialization*). Of course, there's no reason to expect that when W and b are random, the output yield any useful representations.
    - The useless representation is 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
  - 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
  - You'll eventually end up with a model that has a very low loss on its training data: a low mismatch between predictions, y_pred, and expected targets, y_true.
    - Now, the model has "learned" to map its inputs to correct targets
  - Step 1 is straightforward, just I/O. Steps 2 and 3 are merely the application of a handful of tensor operations, so you could implement these steps purely from what you learned in the previous section. The difficult part is step 4: updating the model's weights. Given an individual weight coefficient in the model, how can you compute whether the coefficients should be increased or decreased, and by how much?
    - One naïve solution would be to freeze all weights in the model except the one scalar coefficient being considered, and try different values for this coefficient.
      - Let's say the initial value of the coefficient is 0.3. after the forward pass on a batch of data, the loss of the model on the batch is 0.5. If you change the coefficient's value to 0.35 and return the forward pass, the loss increases to 0.6. But if you lower the coefficient to 0.25, the loss falls to 0.4. In this case, it seems that updating the coefficient by -0.05 would contribute to minimizing the loss. This will have to be repeated for all coefficients in the model
      - But this 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 is a much better approach: **gradient descent**.

- Gradient descent is the optimization technique that powers modern neural networks. Here is the gist of it: all of the functions used in our models (such as dot or +) transform their input in a smooth and continuous way
  - E.g. z = x + y, 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. 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 to the loss of the model on a batch of data.
  - 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.

4.1. Derivative of a tensor operations: the gradient

- The concept of derivation can be applied to any function (with input as multidimension tensors), as long as the surfaces they describe are continuous and smooth. The derivative of a tensor operation is called a ***gradient***.

- Gradients are just the generalization of the concept of derivatives to functions that take tensors as inputs.
  - While for a scalar function, the derivative represents 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. It characterizes how the output of the function varies when its input parameters vary.

- Let's look at an example grounded in machine learning
  - We use *W* to compute a target candidate y_pred, and then compute the loss between the target candidate y_pred and the target y_true:

```
y_pred = dot(W, x)
loss_value = loss(y_pred, y_true)
```

We use the model weights, W, to make a prediction for x.

We estimate how far off the prediction was.

- Now, we'd like to use gradients to figure out how to update *W* so as to make *loss_value* smaller.
  - 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:

```
loss_value = f(W)
```

f describes the curve (or high-dimensional surface) formed by loss values when W varies.

  - 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 when modifying $W0[i, j]$.
    - The 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
  - Concretely, what does $grad(loss_{value}, W0)$ represent?
    - It 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.
  - With a function f(W) of a tensor, you can reduce loss_value by moving W in the opposite direction from the gradient
    - 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

4.2. Stochastic gradient descent

- 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 and check for which of these points the function has the lowest value.

- 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 $grad(f(W), W) = 0$.
    - 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 serval tens of millions.
- Instead, you can apply the four-step algorithm to modify the parameters little by little based on the current loss value for a random batch of data.
    - Because you're dealing with a differentiable function, you can compute its gradient, which gives you an efficient way to implement step 4.
    - If you update the weights in the opposite direction from the gradient, the loss will be a little less every time. Now, the algorithm changes from step 4
        - 1. 2. 3. Compute loss 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* would be a scalar factor modulating the "speed" of the gradient descent process.
- This process 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*).
    - The following figure illustrates what happens in 1D, when the model has only one parameter and you have only one training sample

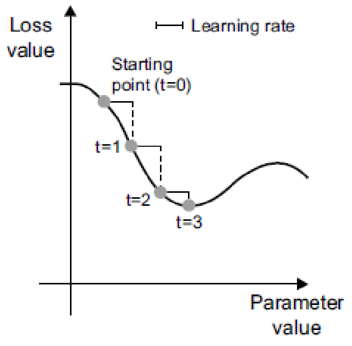


Figure 2.18 SGD down a 1D loss curve (one learnable parameter)

- It's important to pick a reasonable value for the learning rate factor.
    - If it's too small, the decent down the curve will take many iterations, and it could get stuck in a local minimum.
    - If it's too large, your updates may end up taking you to completely random location on the curve

- Some alternatives
    - True SGD: draw a single sample and target at each iteration
    - Batch gradient descent: run every step on all data available
        - Each update would be more accurate, but far more expensive.
    - The efficient compromise between these two extremes is to use mini-batches of reasonable size

- The following figure shows a visualization gradient descent along a 2D loss surface.

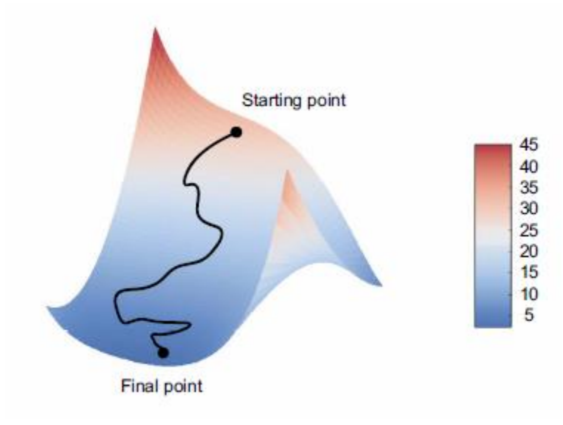


Figure 2.19 Gradient descent down a 2D loss surface (two learnable parameters)

  - ➢ 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 human.
  - ➢ As such, it is important 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.

- 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 example, SGD with momentum, as well as Adagrad, RMSprop, and several others.
  - ➢ Such variants are known as *optimization methods* or ***optimizers***. In particular, the concept of ***momentum***, which is used in many of these variants, deserves your attention.
    - ✧ Momentum addresses two issues with SGD: convergence speed and local minima
  - ➢ Consider the following curve

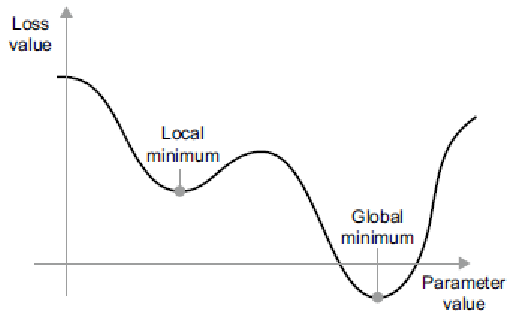
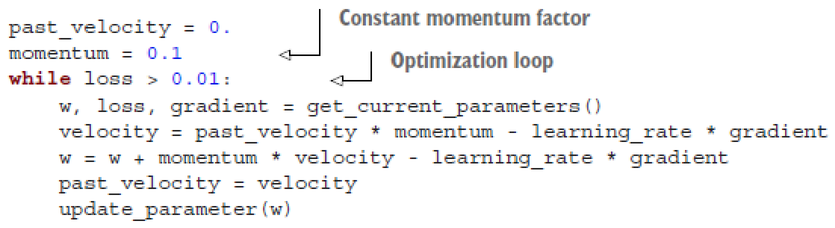


Figure 2.20 A local minimum and a global minimum

    - ✧ 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
  - ➢ 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, such as in the naïve implementation


Constant momentum factor
Optimization loop

```
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)
```

4.3. Chaining derivatives: The Backpropagation algorithm

- In the preceding algorithm, we casually assumed that because a function is differentiable, we can easily compute its gradient. But is this true?
  - ➢ How can we compute the gradient of complex expressions in practice?
  - ➢ In the two-layer model we started the chapter with, how can we get the gradient of the loss with regard to the weights?
  - ➢ That's where the ***Backpropagation algorithm*** comes in

- Chain rule
  - ➢ 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.
    - ✧ For instance, consider the following model, which is parameterized by the variables W1, W2, b1, b2 (used 2 dense layers), involving the atomic operations dot, relu, SoftMax, and +, as well as our loss function, which are 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*.
    - ✧ Consider two functions f and g, as well as the composed function fg such that $fg(x) = f\big(g(x)\big)$, where $g(x) = x1$
    - ✧ The chain rule states that $grad(y,\ x) == grad(y,\ x1) * grad(x1,\ x)$
    - ✧ Consequently, with more intermediate functions, it would look like

```
def fghj(x):
    x1 = j(x)
    x2 = h(x1)
    x3 = g(x2)
    y = f(x3)
    return y

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*.

- Automatic differentiation with computation graphs
  - ➢ A useful way to think about backpropagation is in terms of *computation graphs*.
    - ✧ A computation graph is the data structure at the heart of TensorFlow and the deep learning revolution in general.
    - ✧ It's a directed acyclic graph of operations – in our case, tensor operations.
  - ➢ Computation graphs have been an extremely successful abstraction in computer science because they enable us to *treat computation as data*: a computable expression is encoded as a machine-readable data structure that can be used as the input or output of another program.
    - ✧ For instance, you could imagine a program that receives a computation graph and returns a new computation graph that implements a large-scale distributed version of the same computation – this would mean that you could distribute any computation without having to write the distribution logic yourself.
    - ✧ Or imagine a program that receives a computation graph and can automatically generate the derivative of the expression it represents. It's much easier to do these things if your computation is expressed as an explicit graph data structure rather than, say, lines of ASCII characters in a .py file.

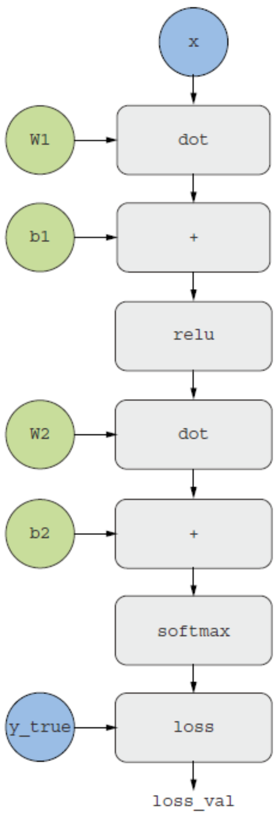


Figure 2.21 The computation graph representation of our two-layer model

- To explain backpropagation clearly, let's look at a really basic example of a computation graph.
  - ➢ Consider a very simple situation where we only have one linear layer and all variables are scalar.
    - ✧ We take two scalar variables W and b, a scalar input x, and apply some operations to them to combine them into an output y.
    - ✧ Finally, we'll apply an absolute value error-loss function.
    - ✧ Since we want to update W and b in a way that will minimize the loss_val, we are interested in computing $grad(loss\_val,\ b)$ and $grad(loss\_val,\ w)$

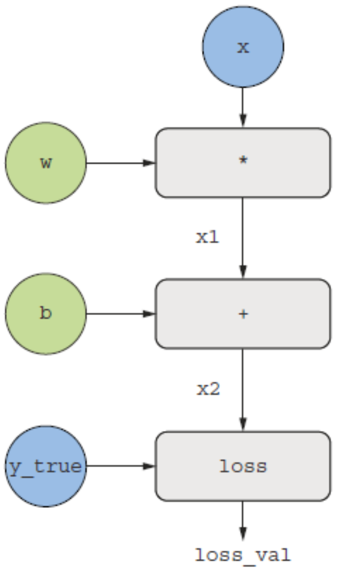


Figure 2.22 A basic example of a computation graph

- 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*

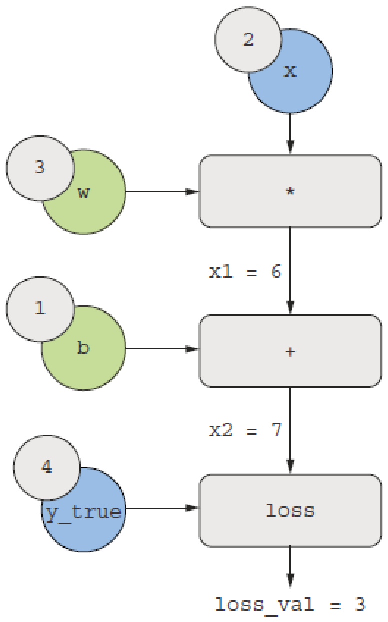


Figure 2.23 Running a forward pass

- 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*.

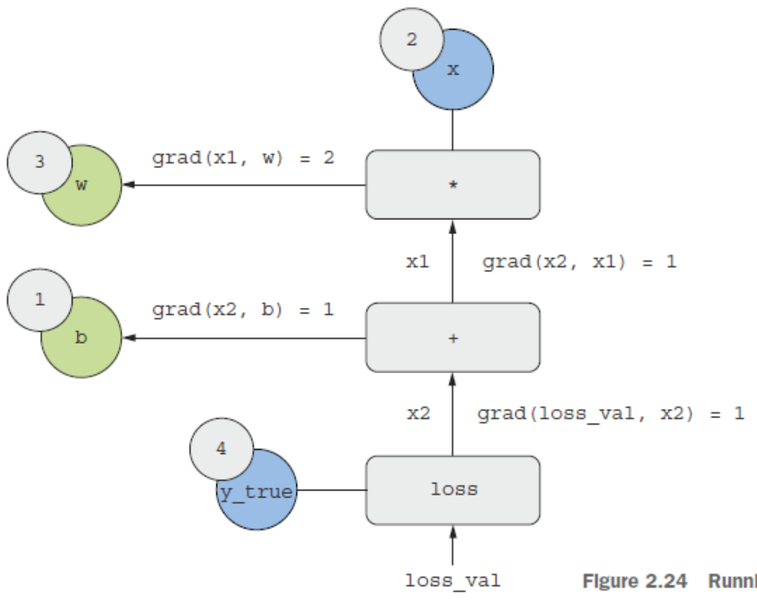


Figure 2.24 Running a backward pass

- The logic is as follows:
  - $grad(loss\_val,\ x2) = 1$, 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
- 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*.
  - For instance, $grad(loss\_val,\ w) = grad(loss\_val,\ x2) * grad(x2,\ x1) * grad(x1,\ w)$

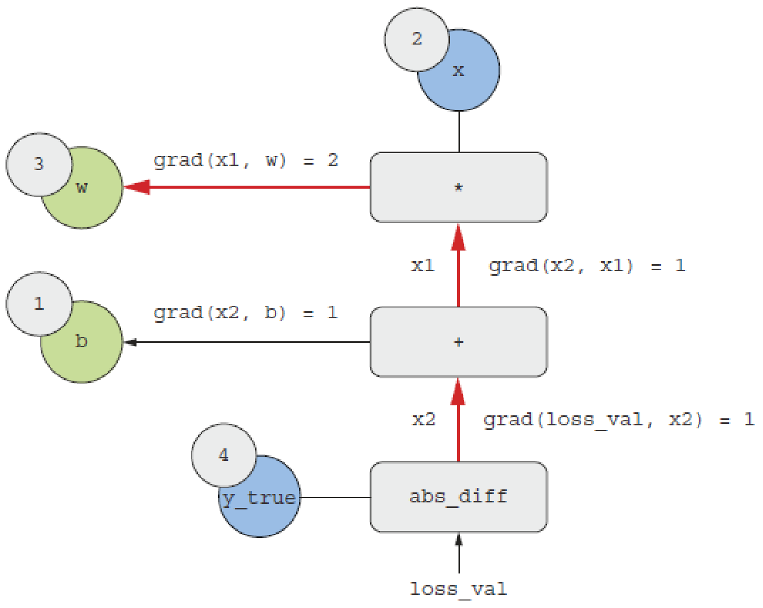


Figure 2.25 Path from `loss_val` to w in the backward graph

- ➢ By applying the chain rule, we have th following
  - `grad(loss_val, w) = 1 * 1 * 2 = 2`
  - `grad(loss_val, b) = 1 * 1 = 1`

- And you just saw backpropagation in action! Backpropagation is simply the application of the chain rule to a computation graph.
  - ➢ 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 contribution of different nodes in a computation graph

- Nowadays, people implement neural networks in modern frameworks that are capable of ***automatic differentiation***, such as TensorFlow.
  - ➢ Automatic differentiation is implemented with the kind of computation graph you've just seen.
  - ➢ 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.

- The **gradient tape** in TensorFlow
  - ➢ The API through which you can leverage TensorFlow's powerful automatic differentiation capabilities is the ***GradientTape***.
    - ✧ It's a python scope that will "record" the tensor operations that run inside it, in the form of a computation graph (sometimes called a "tape").
    - ✧ This graph can then be used to retrieve the gradient of any output with respect to any variable or set of variables (instances of the *tf.Variable* class). A tf.Variable is a specific kind of tensor

meant to hold mutable state – for instance, the weights of a neural network are always tf.Variable instances.

Instantiate a scalar Variable with an initial value of 0.

Open a GradientTape scope.

Inside the scope, apply some tensor operations to our variable.

Use the tape to retrieve the gradient of the output y with respect to our variable x.

```
import tensorflow as tf
x = tf.Variable(0.)
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

➢ The GradientTape works with tensor operations:

```
x = tf.Variable(tf.random.uniform((2, 2)))
with tf.GradientTape() as tape:
    y = 2 * x + 3
grad_of_y_wrt_x = tape.gradient(y, x)
```

Instantiate a Variable with shape (2, 2) and an initial value of all zeros.

grad_of_y_wrt_x is a tensor of shape (2, 2) (like x) describing the curvature of y = 2 * a + 3 around x = [[0, 0], [0, 0]].

➢ It also works with lists of variables

```
W = tf.Variable(tf.random.uniform((2, 2)))
b = tf.Variable(tf.zeros((2,)))
x = tf.random.uniform((2, 2))
with tf.GradientTape() as tape:
    y = tf.matmul(x, W) + b
grad_of_y_wrt_W_and_b = tape.gradient(y, [W, b])
```

matmul is how you say "dot product" in TensorFlow.

grad_of_y_wrt_W_and_b is a list of two tensors with the same shapes as W and b, respectively.

5. Looking back at our first example

- The following figure illustrates what's going on behind the scenes in a neural network

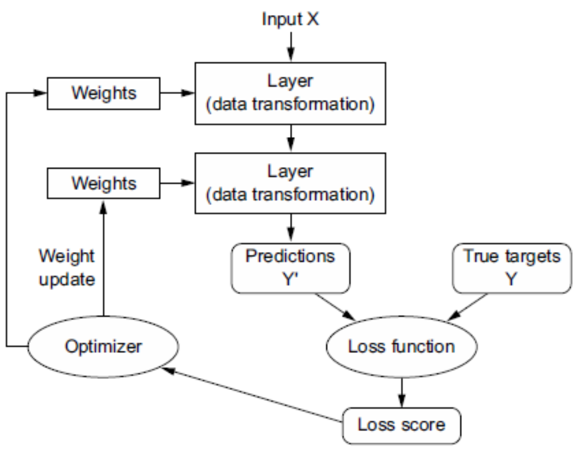


Figure 2.26 Relationship between the network, layers, loss function, and optimizer

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

✧ 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.

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

- Let's go back to the first example in this chapter and review each piece of it in the light of what you've learned since.
  - ➢ Given the input data

    ```
    (train_images, train_labels), (test_images, test_labels) = mnist.load_data()
    train_images = train_images.reshape((60000, 28 * 28))
    train_images = train_images.astype("float32") / 255
    test_images = test_images.reshape((10000, 28 * 28))
    test_images = test_images.astype("float32") / 255
    ```

    - ✧ Recall that input images are stored in NumPy tensors, which are here formatted as *float32* tensors of shape (60000, 784) (training data) and (10000, 784) (test data) respectively.
  - ➢ Our model is given by

    ```
    model = keras.Sequential([
        layers.Dense(512, activation="relu"),
        layers.Dense(10, activation="softmax")
    ])
    ```

    - ✧ Recall that this model consists of a chain of two dense layers, that each layer applies a few simple tensor operations to the input data, and that these operations involve weight tensors.
    - ✧ Weight tensors, which are attributes of the layers, are where the *knowledge* of the model persists.
  - ➢ The compilation step:

    ```
    model.compile(optimizer="rmsprop",
                  loss="sparse_categorical_crossentropy",
                  metrics=["accuracy"])
    ```

    - ✧ Recall that sparse_categorical_crossentropy is the loss function that's used as a feedback signal for learning the weight tensors, and which the training phase will attempt to minimize.
    - ✧ Also recall that the reduction of the loss happens via mini-batch stochastic gradient descent. The exact rules governing a specific use of gradient descent are defined by the *RMSProp* optimizer passed as the first argument
  - ➢ The training loop

    ```
    model.fit(train_images, train_labels, epochs=5, batch_size=128)
    ```

    - ✧ Recall that when you call *fit*: the model will start to iterate on the training data in mini-batches of 128 samples, 5 times over (each iteration over all the training data is called an ***epoch***).
    - ✧ For each batch, the model will compute the gradient of the loss with regard to the weights (using the Backpropagation algorithm, which derives from the chain rule in calculus) and move the weights in the direction that will reduce the value of the loss for this batch.
  - ➢ After these 5 epochs, the model will have performed 2,345 gradient updates (469 per epoch), and the loss of the model will be sufficiently low that the model will be capable of classifying handwritten digits with high accuracy

- At this point, you already know most of what there is to know about neural networks. Let's prove it by reimplementing a simplified version of that first example "from scratch" in TensorFlow, step by step.

5.1. Reimplementing in TensorFlow

- A simple dense class
  - Recall that dense layer implements the following input transformation, where W and b are model parameters, and *activation* is an element-wise function (usually relu, but it would be SoftMax for the last layer):

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

  - Let's implement a simple Python class, NaiveDense, that creates two TensorFlow variables, W and b, and exposes a _call_() method that applies the preceding transformation

```
import tensorflow as tf

class NaiveDense:
    def __init__(self, input_size, output_size, activation):
        self.activation = activation

        w_shape = (input_size, output_size)
        w_initial_value = tf.random.uniform(w_shape, minval=0, maxval=1e-1)
        self.W = tf.Variable(w_initial_value)

        b_shape = (output_size,
        b_initial_value = tf.zeros(b_shape)
        self.b = tf.Variable(b_initial_value)

    def __call__(self, inputs)::
        return self.activation(tf.matmul(inputs, self.W) + self.b)

    @property
    def weights(self):
        return [self.W, self.b]
```

Create a matrix, W, of shape (input_size, output_size), initialized with random values.

Create a vector, b, of shape (output_size,), initialized with zeros.

Apply the forward pass.

Convenience method for retrieving the layer's weights

- A simple sequential class
  - Now, let's create a NaiveSequential class to chain these layers and exposes a _call_() method that simply calls the underlying layers on the inputs, in order. It also features a *weights* property to easily keep track of the layers' parameters

```
class NaiveSequential:
    def __init__(self, layers):
        self.layers = layers

    def __call__(self, inputs):
        x = inputs
        for layer in self.layers:
           x = layer(x)
        return x

    @property
    def weights(self):
       weights = []
       for layer in self.layers:
           weights += layer.weights
       return weights
```

- Using this NaiveDense class and this NaiveSequential class, we can create a mock Keras model:

```
model = NaiveSequential([
    NaiveDense(input_size=28 * 28, output_size=512, activation=tf.nn.relu),
    NaiveDense(input_size=512, output_size=10, activation=tf.nn.softmax)
])
assert len(model.weights) == 4
```

- A batch generator
    - ➢ Next, we need a way to iterate over the MNIST data in mini-batches

```
import math

class BatchGenerator:
    def __init__(self, images, labels, batch_size=128):
        assert len(images) == len(labels)
        self.index = 0
        self.images = images
        self.labels = labels
        self.batch_size = batch_size
        self.num_batches = math.ceil(len(images) / batch_size)

    def next(self):
        images = self.images[self.index : self.index + self.batch_size]
        labels = self.labels[self.index : self.index + self.batch_size]
        self.index += self.batch_size
        return images, labels
```

5.2. Running one training step

- The most difficult part of the process is the "training step": updating the weights of the model after running it on one batch of data. We need to
    - ➢ 1. Compute the predictions of the model for the images in the batch
    - ➢ 2. Compute the loss value for these predictions, given the actual labels.
    - ➢ 3. Compare the gradient of the loss with regard to the model's weights
    - ➢ 4. Move the weights by a small amount in the direction opposite to the gradient.

- To compute the gradient, we will use the TensorFlow GradientTape object we introduced earlier

```
def one_training_step(model, images_batch, labels_batch):
    with tf.GradientTape() as tape:
        predictions = model(images_batch)
        per_sample_losses = tf.keras.losses.sparse_categorical_crossentropy(
            labels_batch, predictions)
        average_loss = tf.reduce_mean(per_sample_losses)
    gradients = tape.gradient(average_loss, model.weights)
    update_weights(gradients, model.weights)
    return average_loss
```

Run the "forward pass" (compute the model's predictions under a GradientTape scope).

Compute the gradient of the loss with regard to the weights. The output gradients is a list where each entry corresponds to a weight from the model.weights list.

Update the weights using the gradients (we will define this function shortly).

- ➢ Recall that the purpose of the "weight update" step is to move the weights by "a bit" in a direction that will reduce the loss on this batch.
    - ✧ The magnitude of the move is determined by the "learning rate", typically a small quantity. The simplest way to implement this update_weights function is to subtract gradient * learning_rate from each weight:

```
learning_rate = 1e-3

def update_weights(gradients, weights):
    for g, w in zip(gradients, weights):
        w.assign_sub(g * learning_rate)
```

assign_sub is the equivalent of -= for TensorFlow variables.

- ➢ In practice, you would almost never implement a weight update step like this by hand. Instead, you would use an *Optimizer* instance from Keras, like this:

```
from tensorflow.keras import optimizers

optimizer = optimizers.SGD(learning_rate=1e-3)

def update_weights(gradients, weights):
    optimizer.apply_gradients(zip(gradients, weights))
```

- Now that our per-batch training step is ready, we can move on to implementing an entire epoch of training.

5.3. The full training loop

- An epoch of training simply consists of repeating the training step for each batch in the training data, and the full training loop is simply the repetition of one epoch:

```
def fit(model, images, labels, epochs, batch_size=128):
    for epoch_counter in range(epochs):
        print(f"Epoch {epoch_counter}")


        batch_generator = BatchGenerator(images, labels)
        for batch_counter in range(batch_generator.num_batches):
            images_batch, labels_batch = batch_generator.next()
            loss = one_training_step(model, images_batch, labels_batch)
            if batch_counter % 100 == 0:
                print(f"loss at batch {batch_counter}: {loss:.2f}")
```

- Let's test drive it

```
from tensorflow.keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28 * 28))
train_images = train_images.astype("float32") / 255
test_images = test_images.reshape((10000, 28 * 28))
test_images = test_images.astype("float32") / 255
```

5.4. Evaluating the model

- We can evaluate the model by taking the argmax of its predictions over the test images, and comparing it to the expected labels:

```
predictions = model(test_images)
predictions = predictions.numpy()
predicted_labels = np.argmax(predictions, axis=1)
matches = predicted_labels == test_labels
print(f"accuracy: {matches.mean():.2f}")
```

Calling .numpy() on a TensorFlow tensor converts it to a NumPy tensor.

- All done! It's quite a bit of work to do "by hand" what you can do in a few lines of Kears code.
  - ➢ But it is important to understand what goes on inside a neural network when you call *fit()*.

- Having this low-level mental model of what your code is doing behind the scenes will make you better able to leverage the high-level features of the Keras API

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 transformation. 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 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

- 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 gradient value. This is called backpropagation.

- Two key concepts you'll see frequently in th future chapters are loss and optimizers. These are the two things 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

Appendix: defining a class in Python

- Python is an object oriented programming language, almost everything in Python is an object, with its properties and methods.
    - ➢ A Class is like an object constructor, or a "blueprint" for creating objects

Constructing a Class

- To begin, in order to create a class object, you need to use the key word *class*, and you can start defining first function.
    - ➢ Usually, when creating a class, you will have to define a function called _init_ with *self* as the initial argument

```
class CreateProfile:
    def __init__(self):
```

- ➢ The **_init_** function is called when a class if instantiated (when you declare the class, which can happen either by itself or by assigning it to a variable, considering the following examples)

```
CreateProfile()

# OR

profile = CreateProfile()
```

- ➢ When the class object is instantiated, we implicitly called the _init_ function.
- ➢ Any arguments within the _init_ function will also be the same arguments when instantiating the class object
    - ✧ These initial arguments can be the data we wish to manipulate throughout the class object.
    - ✧ But in regards to the *self* argument, it will not be necessary to replace when instantiating the class object.
- ➢ The **self** argument is an implicit argument that will always be called when instantiating the class or when you use a custom function within that class.
    - ✧ The self refers to the object it is manipulating.
    - ✧ Consider the following continuation of our class definition

```
class CreateProfile:
    def __init__(self,
                 dataset=None,
                 profile=None):


        self.dataset = dataset
        self.profile = profile
```

- ➢ We defined some additional arguments after self, in order for them to be used throughout the class, we must assign them to the object or self that we will manipulate with functions later on.
- ➢ Note that, when assigning values in the _init_ function, we are effectively creating **class attributes**.

- They can be called upon after instantiating the class and they will return whatever variable we assigned to the object self

```
# Instantiating the class
new_profile = CreateProfile()


# Calling the class attribute
new_profile.dataset
```

- Functions that we create within a class are called *methods*.
    - Functions and methods are essentially the same thing but in the context of class objects, functions are referred to as methods.

```
    # Here we add another function
    def enter_info(self):
        # Code goes here
```

```
# Running each method after having ran enter_info()
new_profile.add_profile_to_dataset()


new_profile.vect_text()


new_profile.scale_profile()


new_profile.format_profile()
```

- Consider in this example that, attributes with the same name as the methods are defined, and we can access those by

```
# Running each attribute to check their results
new_profile.combined_df.tail(5)


new_profile.vectorized_text


new_profile.scaled_profile


new_profile.formatted_profile
```