بنام خداوند جان و خرد

Mathematical Building Blocks of Neural Networks

Mohammad Taher Pilehvar

Introduction to Artificial Intelligence, 97

http://iust-courses.github.io/ai97/

Today

- A first example of a neural network!
- What are tensors?
- What is *gradient descent*?
- How does *backpropagation* work?

Our first network: handwritten recognition

- Digit classification
  - A 28x28 image into 10 categories (0 to 9)

- MNIST
  - An old classic dataset (1980s)
  - 60K training examples + 10K test

libraries

Importing MNIST data

- Images come preloaded with Keras, as Numpy arrays.

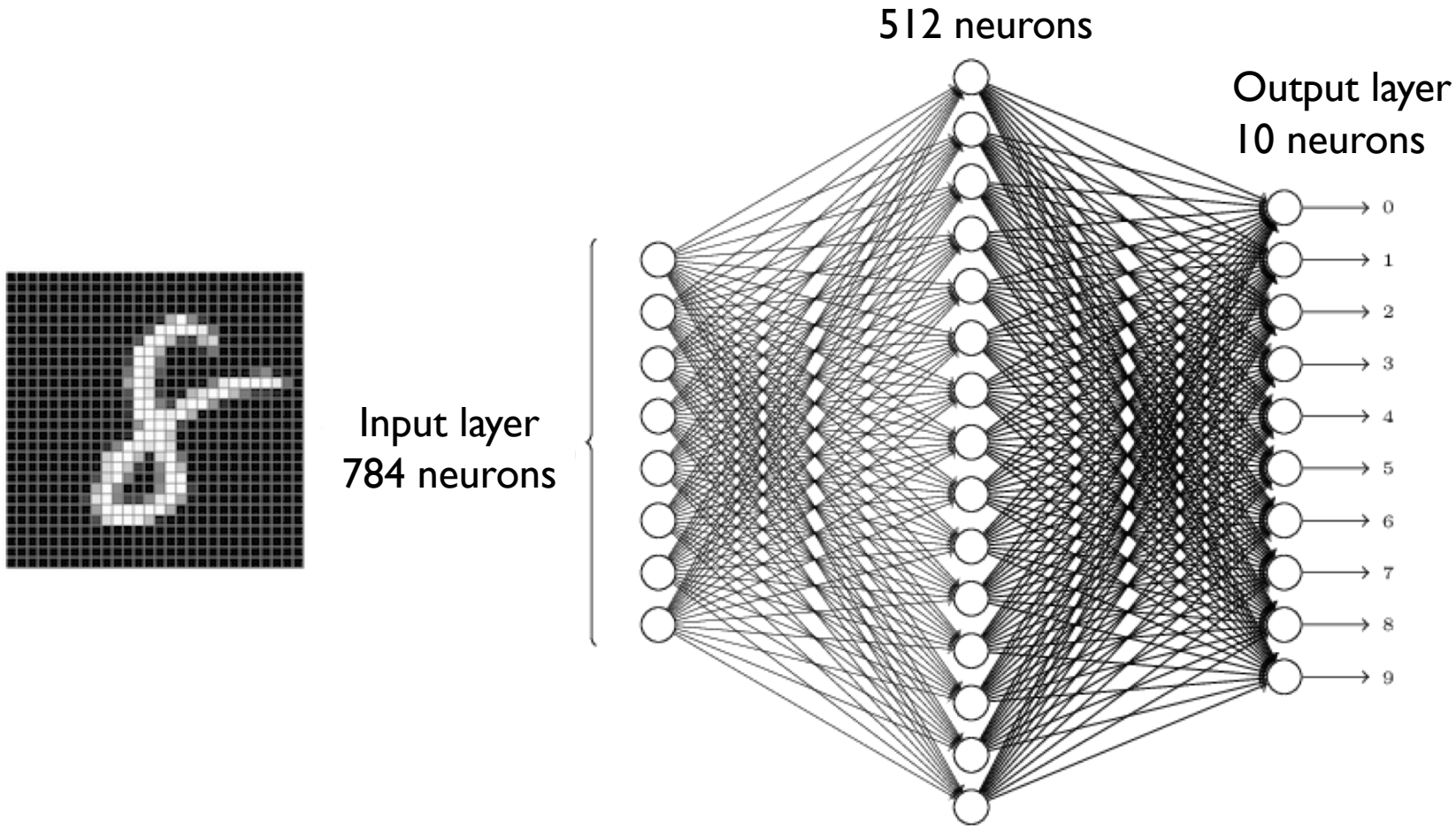
```
from keras.datasets import mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

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

First network implementation in Keras

```
from keras import models
from keras import layers
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

- Two `Dense` (fully connected) layers.
- Layers extract representations out of the data fed into them.
- Chaining together simple layers that will implement a form of progressive *data distillation*.

Dense: MLP


512 neurons
Output layer
10 neurons
Input layer
784 neurons
0
1
2
3
4
5
6
7
8
9

First network implementation in Keras

But we need more:

- A *loss function* – to measure performance on the training data, to steer in the right direction
- An *optimizer* – to update the network
- A metric – how is the performance measured?

```
network.compile(optimizer='rmsprop',
                loss='categorical_crossentropy',
                metrics=['accuracy'])
```

First network implementation in Keras

Some data preprocessing

- Reshape into the shape the network expects, while scaling to [0, 1]

```
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
```

- Categorically encode the labels

```
from keras.utils import to_categorical
train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)
```

jupyter

First network implementation in Keras

We are ready to train!

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

Epoch 1/5
60000/60000 [==============================] - 9s - loss: 0.2524 - acc: 0.9273
Epoch 2/5
51328/60000 [========================>.....] - ETA: 1s - loss: 0.1035 - acc:
0.9692
```

And we can test the model:

```
>>> test_loss, test_acc = network.evaluate(test_images, test_labels)
>>> print('test_acc:', test_acc)
test_acc: 0.9785
```

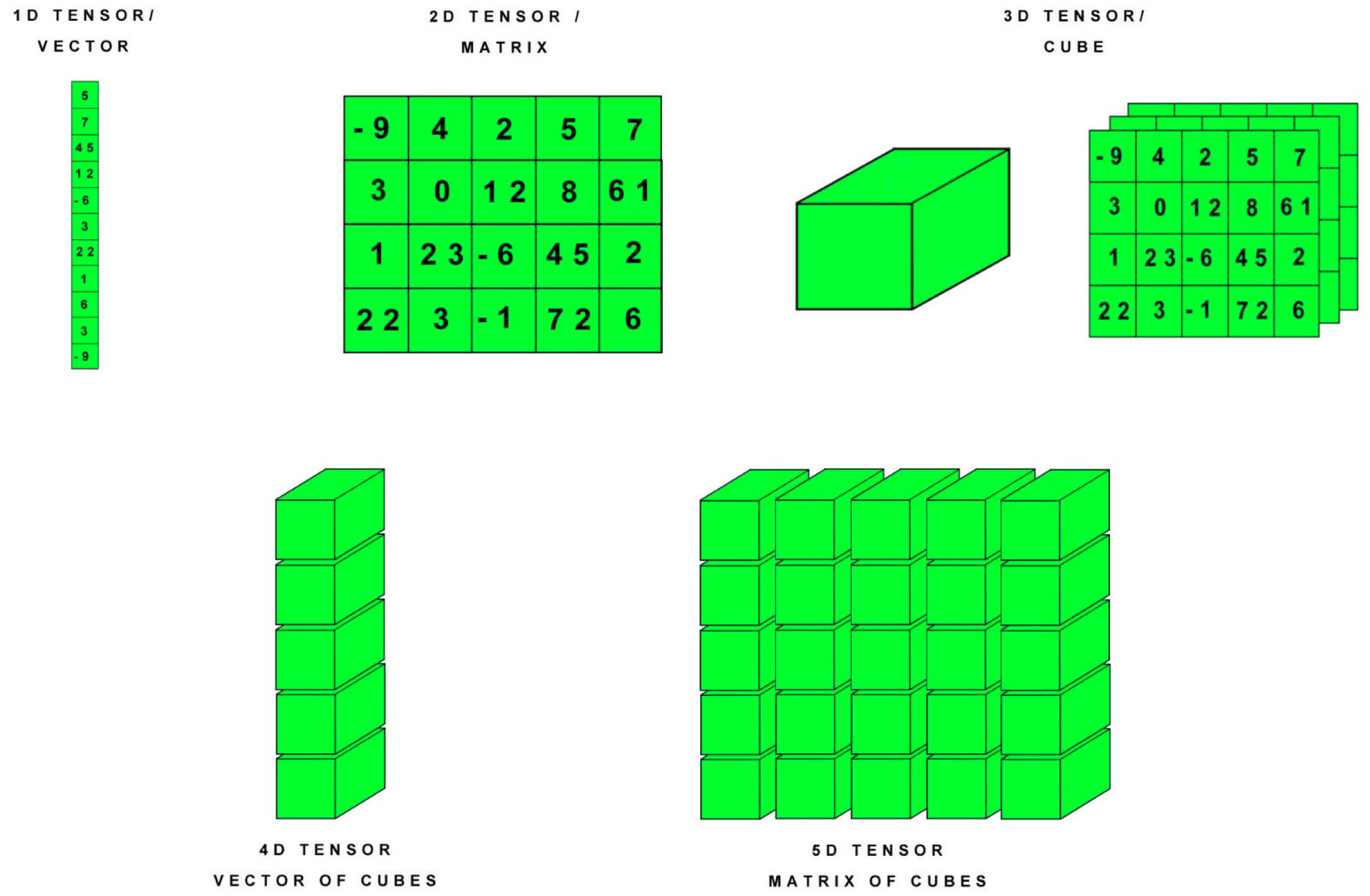
What are Tensors?

All current machine-learning systems use tensors as their basic data structure

- A container for data - almost always numerical data
- Matrices are 2D tensors - Tensors are a generalization of matrices to an arbitrary number of dimensions

- In the context of tensors, a *dimension* is often called an *axis*

What are tensors?



From hackernoon.com

Scalars and vectors

- A tensor that contains only one number is called a *scalar* (or scalar tensor, or 0-dimensional tensor, or 0D tensor).

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

- An array of numbers is called a vector, or 1D tensor.

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

jupyter

Matrices

- An array of *vectors* is a *matrix*, or 2D *tensor*.

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

- The entries from the first axis are called the *rows*, and the entries from the second axis are called the *columns*.

jupyter

3D tensors

- If you pack such matrices in a new array, you obtain a 3D tensor

```
>>> 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
```

Key attributes of a tensor

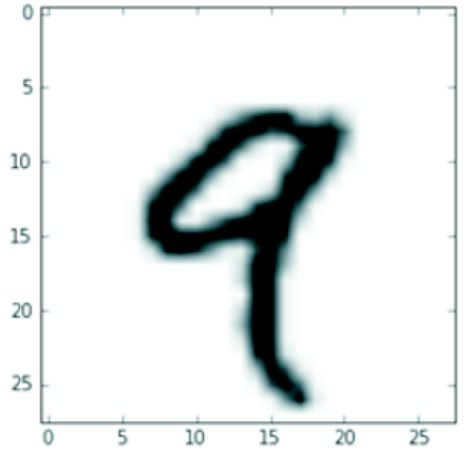
- Number of axes (rank)
  - Tensor's ndim in Python libraries such as Numpy

- Shape
  - A tuple of integers that describes how many dimensions the tensor has along each axis, for instance `(3, 5)` or `(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
  - `float32`, `uint8`, `float64`, and so on
  - Sometimes, `char`

In our example

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

>>> print(train_images.ndim)
3

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

>>> print(train_images.dtype)
uint8

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



Manipulating tensors in Numpy

- Selecting specific elements in a tensor is called *tensor slicing*.
  - Select digits #10 to #100 (#100 isn't included) and put them in an array of shape (90, 28, 28):

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

  - Equivalently (: is equivalent to selecting the entire axis)

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

Manipulating tensors in Numpy

- Select 14 X 14 pixels in the bottom-right corner of all images

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

- Crop the images to patches of 14 X 14 pixels centered in the middle
  - Much like negative indices in Python lists, they indicate a position relative to the end of the current axis

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

Data *batch*

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 MNIST digits, with batch size of 128

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

- And the next batch:

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

- And the n^{th} batch:

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

Sample dimension

- In general, the first axis (axis 0, because indexing starts at 0) in all data tensors you'll come across in deep learning will be the samples axis (sometimes called the samples dimension).

Data tensors

- Vector data - 2D tensors of shape `(samples, features)`
- Timeseries data or sequence data - 3D tensors of shape `(samples, timesteps, features)`
- Images - 4D tensors of shape `(samples, height, width, channels)` or `(samples, channels, height, width)`
- Video - 5D tensors of shape `(samples, frames, height, width, channels)` or`(samples, frames, channels, height, width)`

Vector data

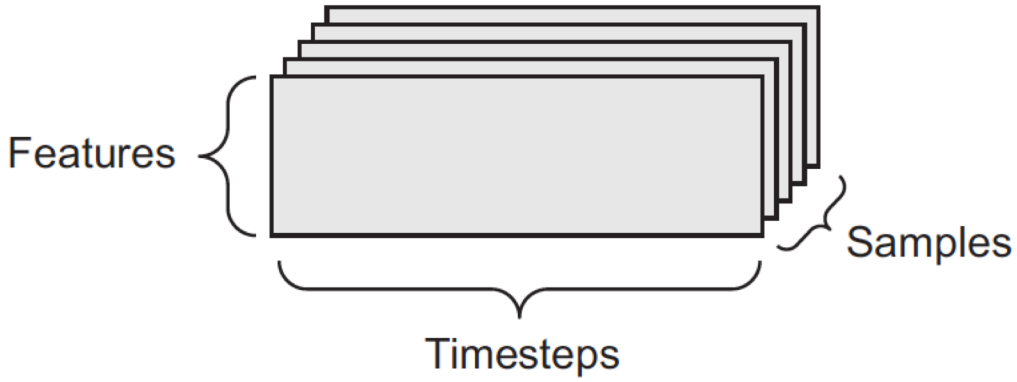
- Each single data point can be encoded as a vector
  - Thus a batch of data will be encoded as a 2D tensor
  - The first axis is the *samples axis* and the second axis is the *features axis*

- Examples:
  - A dataset of 100,000 people, where we consider each person's *age*, *post code*, and *income:* a 2D 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): a 2D tensor of (500, 20000).

Timeseries and sequence data

- Whenever time matters in your data (or the notion of sequence order), it makes sense to store it in a 3D tensor with an explicit time axis.



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

Image data

Images typically have three dimensions: height, width, and color depth.

- A batch of 128 grayscale images of size 256 X 256 could thus be stored in a tensor of shape `(128, 256, 256, 1)`

- A batch of 128 color images could be stored in a tensor of shape `(128, 256, 256, 3)`

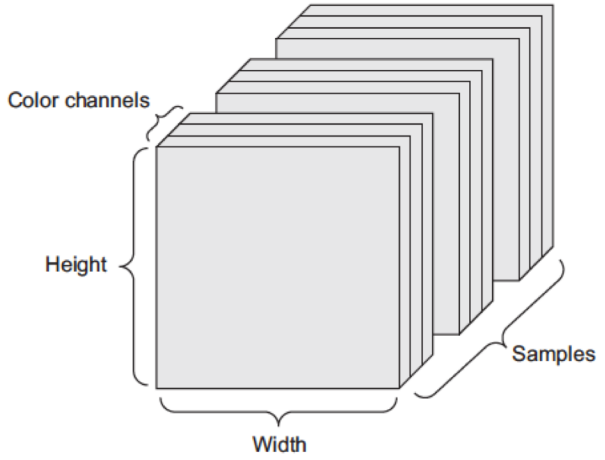


Image data

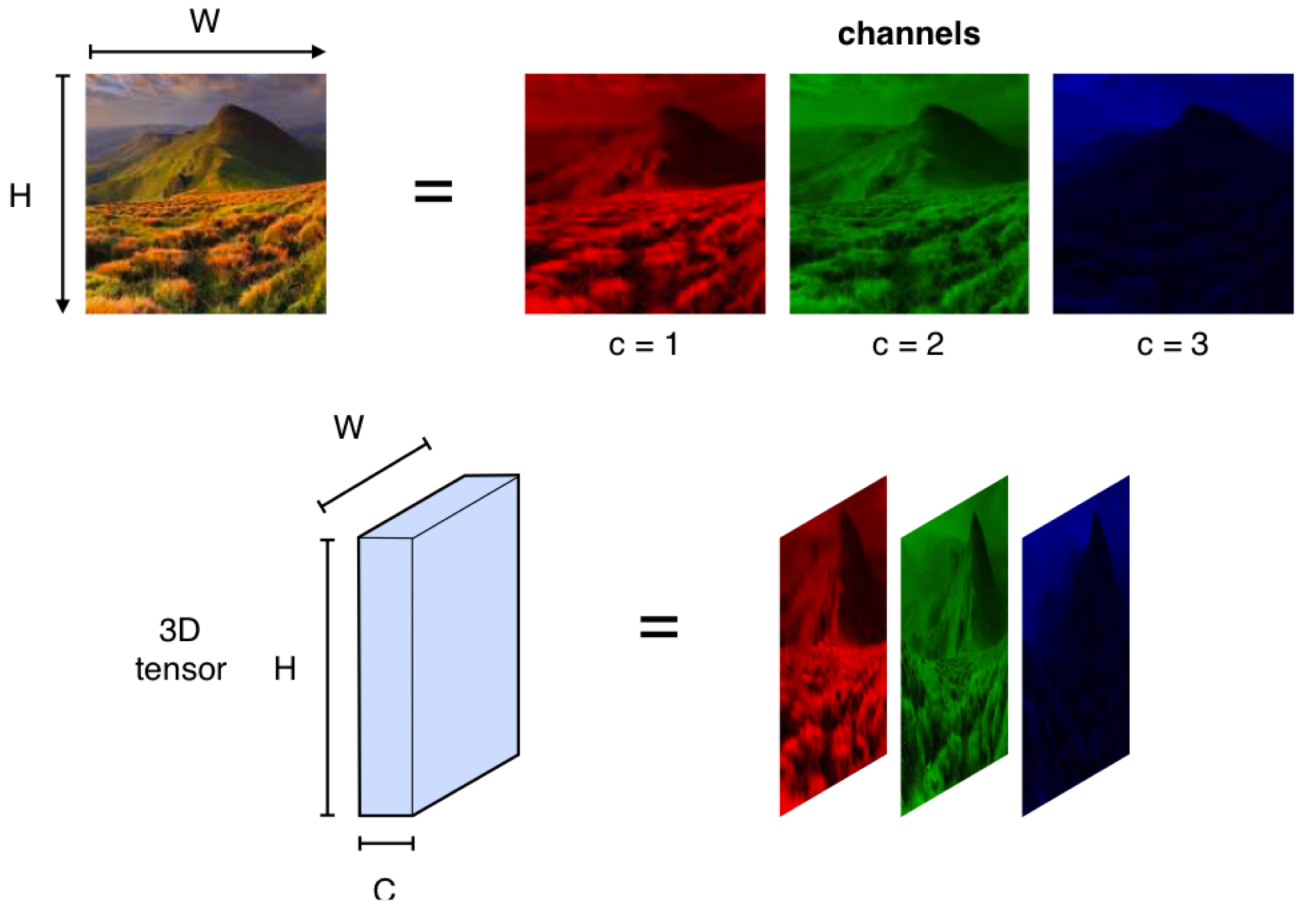


Illustration by Marc Lelarge

Video data

A sequence of frames, each frame being a color image.

```
(samples, frames, height, width, color_depth)
```

- For instance, a batch of 4 YouTube video clips of 60-second, 144 X 256 sampled at 4 frames per second would have 240 frames.

```
(4, 240, 144, 256, 3)
```

- That's a total of 106,168,320 values! If the dtype of the tensor was float32, then each value would be stored in 32 bits, so the tensor would represent 405 MB.

Tensor operations

All transformations learned by deep neural networks can be reduced to a handful of tensor operations applied to tensors of numeric data

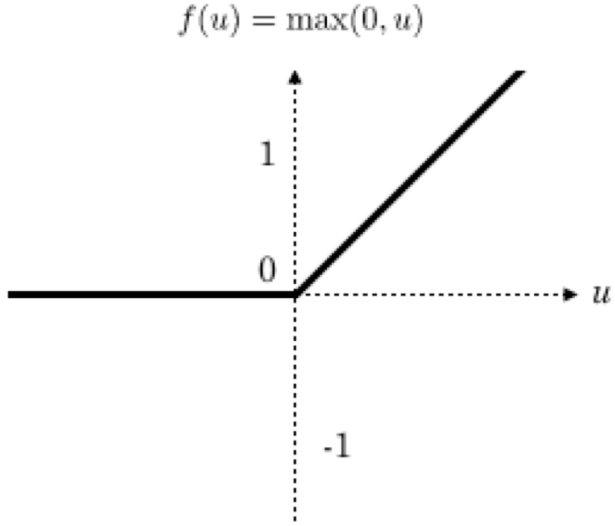
From our example:

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

This layer can be interpreted as a function, which takes as input a 2D tensor and returns another 2D tensor:

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

Tensor operations




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

- We have three operations:
  - Dot product (`dot`) between `input` and `W` tensors
  - An addition between the resulting 2D tensor and a vector `b`
  - And finally a `relu` operation:
    - `relu(x) = max(x,0)`

Element-wise operations

- Addition and relu are element-wise operations
  - Highly parallelizable

```
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
```

```
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
```

- On the same principle, you can do element-wise multiplication, subtraction, and so on.

Broadcasting

- In the `Dense` layer of our example, we added a 2D tensor with a vector. How?

- When possible, and if there's no ambiguity, the smaller tensor will be broadcasted to match the shape of the larger tensor. Broadcasting 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.

Broadcasting

Example:

- Consider `X` with shape `(32, 10)` and `y` with shape `(10,)`. First, we add an empty first axis to `y`, whose shape becomes `(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)`. At this point, we can proceed to add `X` and `Y`, because they have the same shape.

- Broadcasting automatically detects the dimensions (The output `z` has shape `(64, 3, 32, 10)` like `x`):

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

Tensor dot

Also called a tensor product

- (not to be confused with an elementwise product)

```
import numpy as np
z = np.dot(x, y)
```

- The output is a scalar
- Vectors with the same number of elements are compatible for a dot product

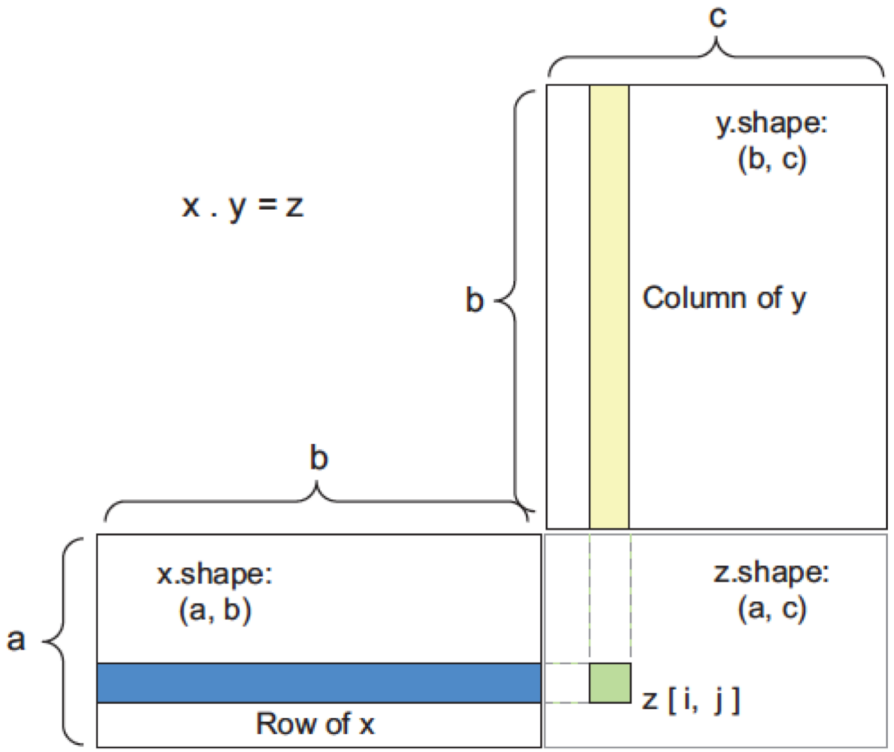
Tensor dot

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])
```



Tensor reshaping

- In our example, we used it when we preprocessed the digits data before feeding it into our network

```
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.

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

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

Tensor operations: Geometric interpretation

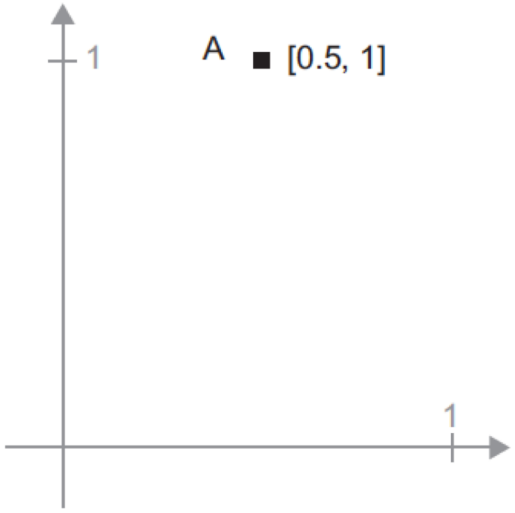


Figure 2.6 A point in a 2D space

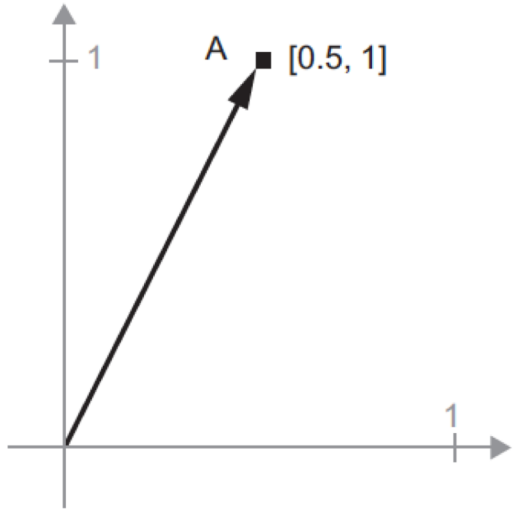


Figure 2.7 A point in a 2D space pictured as an arrow

Tensor operations: Geometric interpretation

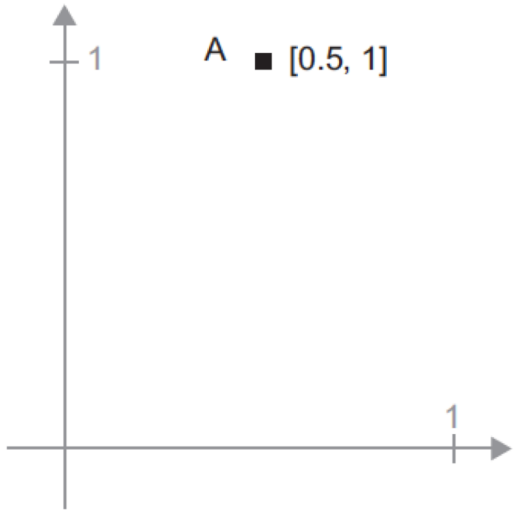


Figure 2.6 A point in a 2D space

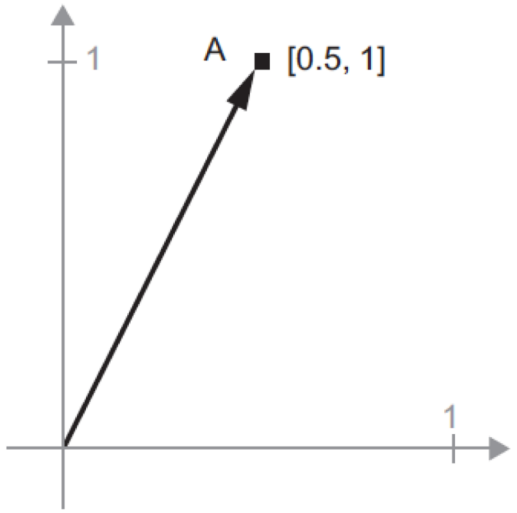


Figure 2.7 A point in a 2D space pictured as an arrow

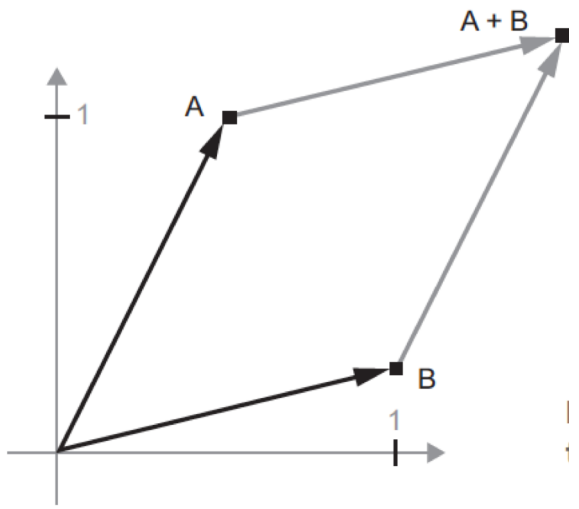


Figure 2.8 Geometric interpretation of the sum of two vectors

Gardient-based optimization

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

`W` and `b` are *weights* or *trainable parameters* of the layer

- These weights contain the information learned by the network from exposure to training data.

Initially, these are filled with random values (*random initialization*)

But, we gradually adjust these weights based on the feedback signal

Training loop

This gradual adjustment, also called *training*, is basically the learning that machine learning is all about.

1. Draw a batch of training samples `x` and corresponding targets `y`.
2. Run the network on `x` (a step called the *forward pass*) to obtain predictions `y_pred`.
3. Compute the *loss* of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Update all weights of the network in a way that slightly reduces the loss on this batch.

Training loop

But, how to update the weights of the network?

1. Draw a batch of training samples `x` and corresponding targets `y`.
2. Run the network on `x` (a step called the *forward pass*) to obtain predictions `y_pred`.
3. Compute the *loss* of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Update all weights of the network in a way that slightly reduces the loss on this batch.

How to adjust weights?

One solution:

- Freeze all weights in the network except the one scalar coefficient being considered, and try different values for this coefficient.
- Slightly change the value and monitor the *loss*.

- Horribly inefficient!

- A much better solution:
  - All operations in the network are *differentiable*
  - Compute the *gradient* of *loss* with respect to coefficients

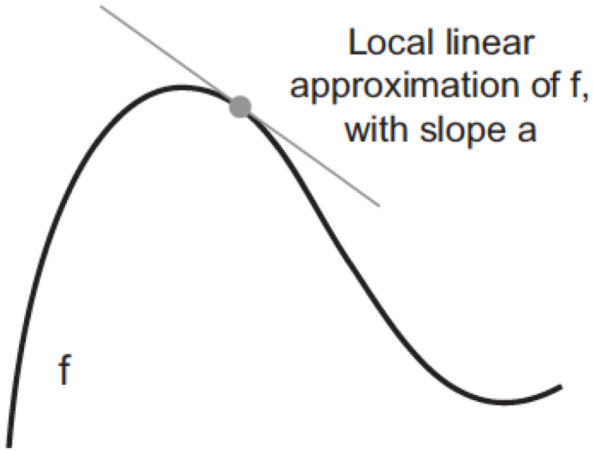
Some basics: derivative

Consider a continuous, smooth function `f(x) = y`

- We'd like to see how much `f(x)` changes with small changes in `x` around point `p`

- This can be computed as the slope of `f(x)` around point `p`

- The slope a is called the *derivative* of `f` in `p`



Gradient

A *gradient* is the derivative of a tensor operation.

Stochastic gradient descent (SGD)

- Given a differentiable function, it's theoretically possible to find its minimum analytically: where derivative is 0.
  - Find the mimimum point where gradient is 0

- But, in neural networks there are thousands (often millions) are parameters
  - Impossible to solve analytically

- Instead, we can use SGD, an iterative algorithm
  - Change parameters little by little to minimize the *loss*

Stochastic gradient descent (SGD)

Mini-batch stochastic gradient descent

1. Draw a batch of training samples `x` and corresponding targets `y`.
2. Run the network on `x` to obtain predictions `y_pred`.
3. Compute the *loss* of the network on the batch, a measure of the mismatch between `y_pred` and `y`.
4. Compute the gradient of the *loss* with regard to the network's parameters (a backward pass).
5. Move the parameters a little in the opposite direction from the gradient – for example `W -= step * gradient` – thus reducing the loss on the batch a bit.

Stochastic gradient descent (SGD)

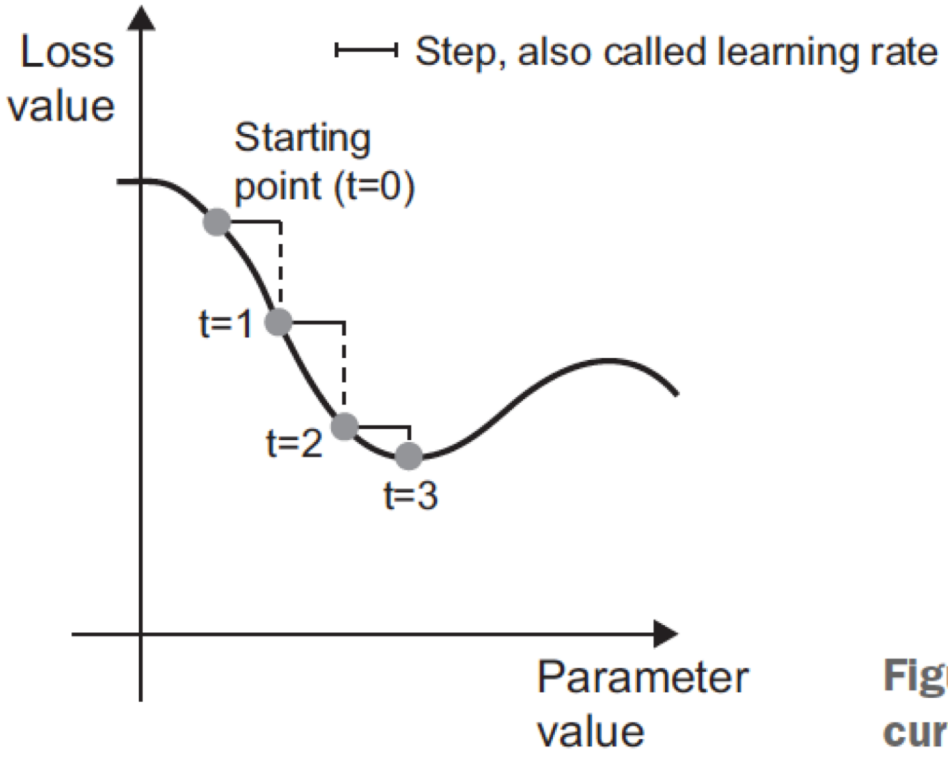


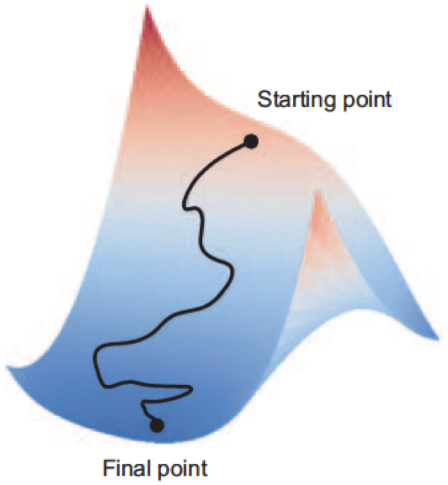
Figure 2.11 SGD down a 1D loss curve (one learnable parameter)

Stochastic gradient descent (SGD)

- Step (learning rate)
  - If too small, the descent down the curve will take many iterations, and it could get stuck in a local minimum.
  - If too large, your updates may end up taking you to completely random locations on the curve.

- True SGD: batch contains only one sample

- Batch SGD: batch contains the whole data



Other optimization methods

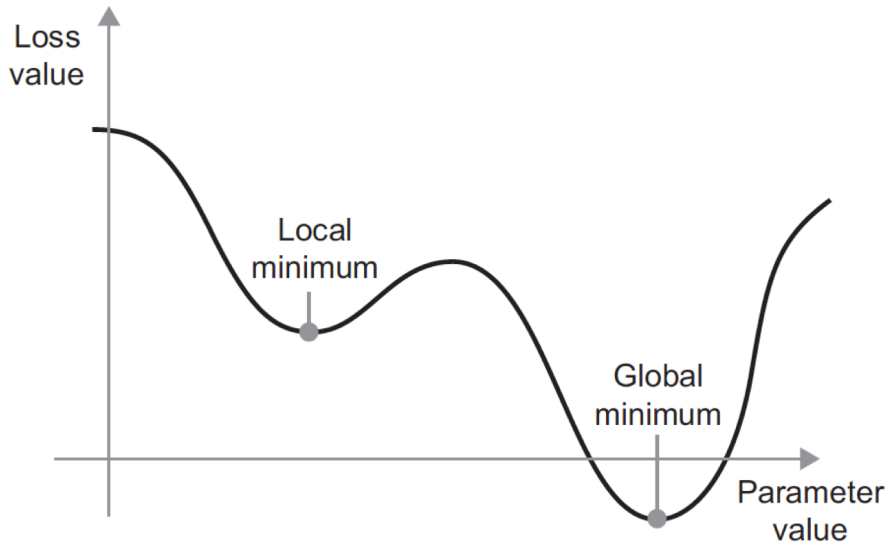
- Take into account previous weight updates when computing the next weight update, rather than just looking at the current value of the gradients.

- *Momentum*:

  Momentum addresses two issues with SGD: convergence speed and local minima.

Momentum

- Inspired from physics



- Consider 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
- Current slope value (current acceleration) + current velocity (resulting from past acceleration)

Backpropagation

- A neural network function consists of many tensor operations chained together, each of which has a simple, known derivative
  - For instance, this is a network `f` composed of three tensor operations, `a`, `b`, and `c`, with weight matrices `W1`, `W2`, and `W3`:

```
f(W1, W2, W3) = a(W1, b(W2, c(W3)))
```

- Chain rule:
  - derivative of `f(g(x))` can be computed as `f'(g(x)) * g'(x)`

- Backpropagation starts with the final loss value and works backward from the top layers to the bottom layers, applying the chain rule to compute the contribution that each parameter had in the loss value.

```
(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
```

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.

```
(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
```

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

this network 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.

```
(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
```

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

```
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

`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.

```
(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
```

```
network = models.Sequential()
network.add(layers.Dense(512, activation='relu', input_shape=(28 * 28,)))
network.add(layers.Dense(10, activation='softmax'))
```

```
network.compile(optimizer='rmsprop',
loss='categorical_crossentropy',
metrics=['accuracy'])
```

The network 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).

At each iteration, the network will compute the gradients of the weights with regard to the loss on the batch, and update the weights accordingly.

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

Questions?

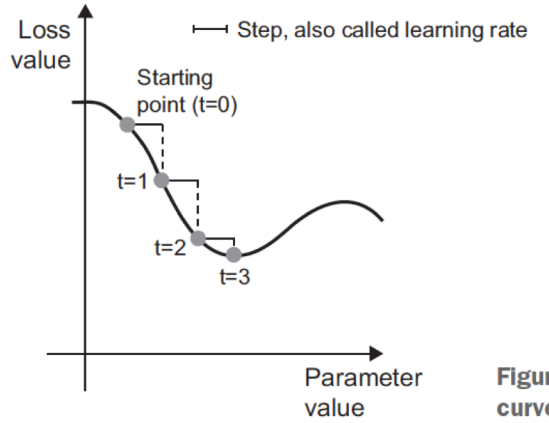


Figure 2.11 SGD down a 1D loss curve (one learnable parameter)