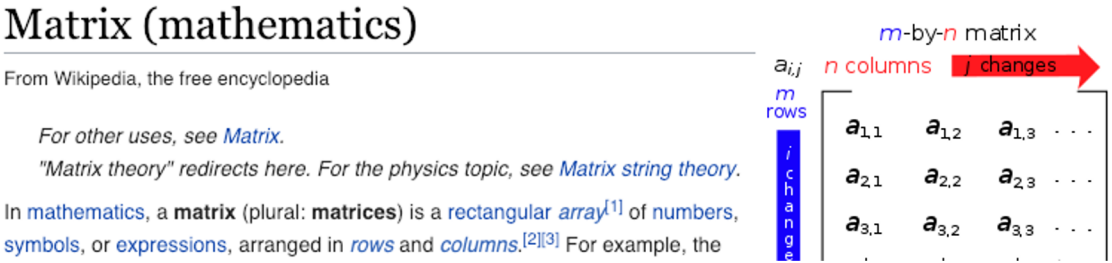

# Data representations and tensor operations

# 2.2 Data representations for neural networks

- *Tensors* are multidimensional Numpy arrays
- A tensor is a container for data—almost always numerical data
  - it's a container for numbers
- All current machine-learning systems use tensors as their basic data structure
- Matrices are 2D tensors
  - Tensors are a generalization of matrices to an arbitrary number of dimensions

# Matrix vs Tensor

- A matrix is a grid of n × m (say, 3 × 3) numbers surrounded by brackets
  - We can add and subtract matrices of the same size, multiply one matrix with another as long as the sizes are compatible ((n × m) × (m × p) = n × p), and multiply an entire matrix by a constant
  - A vector is a matrix with just one row or column

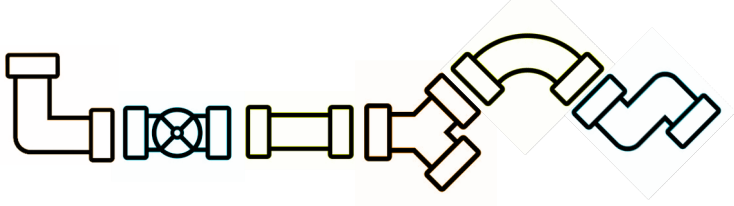



- A tensor is a generalized matrix
  - It could be a 1-D matrix (a vector is actually such a tensor)
  - A 3-D matrix (something like a cube of numbers)
  - Or even a 0-D matrix (a single number),
  - Or a higher dimensional structure that is harder to visualize

# Matrix vs Tensor

- It is not just that tensors are a generalization of matrices!
- A tensor is a mathematical entity that lives in a structure and interacts with other mathematical entities
  - If we transform the other entities in the structure in a regular way, then the tensor must obey a related transformation rule

Structure can be the “number of dimensions of each block and the data type”



- This “dynamical” property is the key that distinguishes a tensor from a mere matrix
  - It’s a team player whose numerical values shift around along with those of its teammates when a transformation is introduced that affects all of them

# 2.2.5 A Tensor in “Tensorflow”

- A `tf.Tensor` has the following properties:
  - Number of axes (rank)
  - Data type (float32, int32, or string, for example)
  - Shape
- The **rank** of a tf.Tensor object is its number of dimensions
  - Synonyms for rank: **order** or **degree** or **n-dimension**
- The **shape** of a tensor
  - is the number of elements in each dimension
- Tensors have a **data type**
  - It is not possible to have a tf.Tensor with more than one data type

| Rank | Shape | Dimension number | Example |
|---|---|---|---|
| 0 | [] | 0-D | A 0-D tensor. A scalar. |
| 1 | [D0] | 1-D | A 1-D tensor with shape [5]. |
| 2 | [D0, D1] | 2-D | A 2-D tensor with shape [3, 4]. |
| 3 | [D0, D1, D2] | 3-D | A 3-D tensor with shape [1, 4, 3]. |
| n | [D0, D1, ... Dn-1] | n-D | A tensor with shape [D0, D1, ... Dn-1]. |

| Rank | Math entity |
|---|---|
| 0 | Scalar (magnitude only) |
| 1 | Vector (magnitude and direction) |
| 2 | Matrix (table of numbers) |
| 3 | 3-Tensor (cube of numbers) |
| n | n-Tensor (you get the idea) |

- `tf.float16` : 16-bit half-precision floating-point.
- `tf.float32` : 32-bit single-precision floating-point.
- `tf.float64` : 64-bit double-precision floating-point.
- `tf.bfloat16` : 16-bit truncated floating-point.
- `tf.complex64` : 64-bit single-precision complex.
- `tf.complex128` : 128-bit double-precision complex.
- `tf.int8` : 8-bit signed integer.
- `tf.uint8` : 8-bit unsigned integer.
- `tf.uint16` : 16-bit unsigned integer.
- `tf.uint32` : 32-bit unsigned integer.
- `tf.uint64` : 64-bit unsigned integer.
- `tf.int16` : 16-bit signed integer.
- `tf.int32` : 32-bit signed integer.
- `tf.int64` : 64-bit signed integer.
- `tf.bool` : Boolean.
- `tf.string` : String.
- `tf.qint8` : Quantized 8-bit signed integer.
- `tf.quint8` : Quantized 8-bit unsigned integer.
- `tf.qint16` : Quantized 16-bit signed integer.
- `tf.quint16` : Quantized 16-bit unsigned integer.
- `tf.qint32` : Quantized 32-bit signed integer.
- `tf.resource` : Handle to a mutable resource.
- `tf.variant` : Values of arbitrary types.

# 2.2.1 Scalars (0D tensors)

- A tensor that contains only one number is called a scalar (or scalar tensor, or 0-dimensional tensor, or 0D tensor)
  - In Numpy, a float32 or float64 number is a scalar tensor (or scalar array)
- You can display the number of axes of a Numpy tensor via the ndim attribute
  - a scalar tensor has 0 axes (ndim == 0)
  - 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
```

# 2.2.2 Vectors (1D tensors)

- An array of numbers is called a vector, or 1D tensor
- A 1D tensor is said to have exactly one axis

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

# 2.2.3 Matrics (2D tensors)

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

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

- Example:
  - Pima Indian Diabetes Dataset (a tabular data)
  - https://www.kaggle.com/uciml/pima-indians-diabetes-database

| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 6 | 148 | 72 | 35 | 0 | 33.6 | 0.627 | 50 | 1 |
| 1 | 1 | 85 | 66 | 29 | 0 | 26.6 | 0.351 | 31 | 0 |
| 2 | 8 | 183 | 64 | 0 | 0 | 23.3 | 0.672 | 32 | 1 |
| 3 | 1 | 89 | 66 | 23 | 94 | 28.1 | 0.167 | 21 | 0 |
| 4 | 0 | 137 | 40 | 35 | 168 | 43.1 | 2.288 | 33 | 1 |

# 2.2.4 3D Tensors and higher-dimensional tensors

- If you pack 2D matrices in a new array, you obtain a 3D tensor
  - you can visually interpret as a cube of numbers

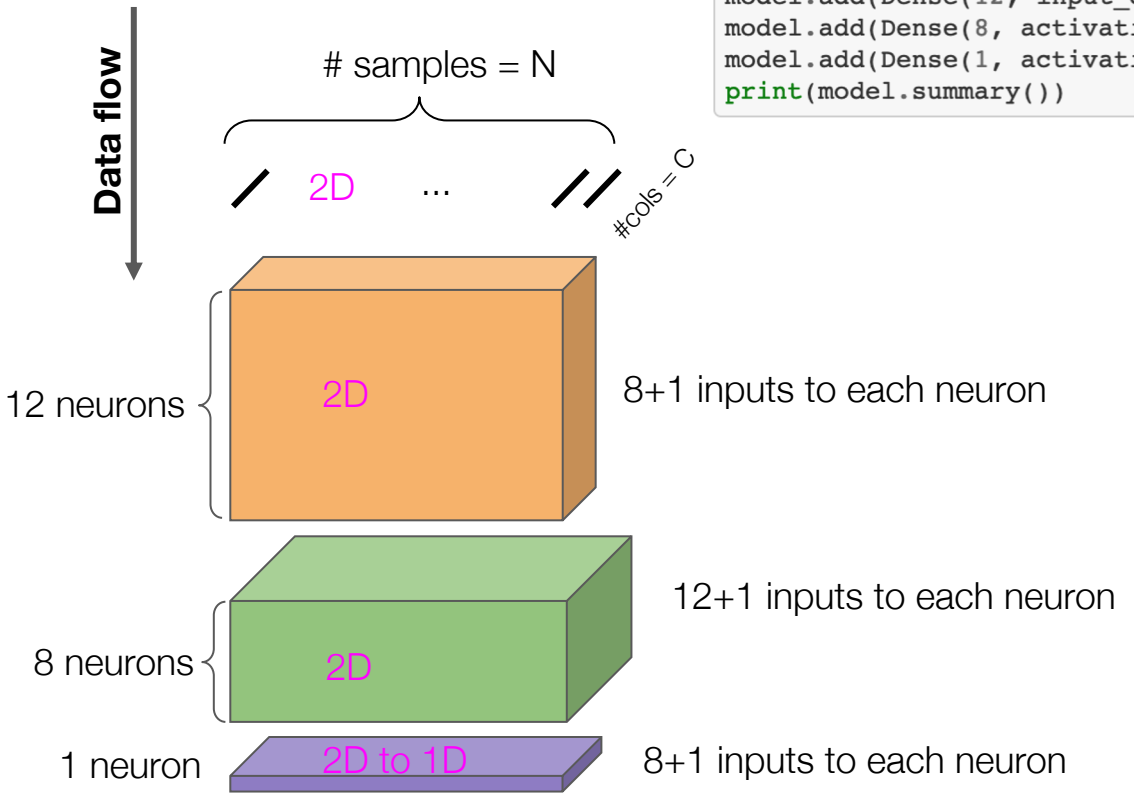
```
>>> x = np.array([[[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3, 35, 1],
                   [7, 80, 4, 36, 2]],
                  [[5, 78, 2, 34, 0],
                   [6, 79, 3,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

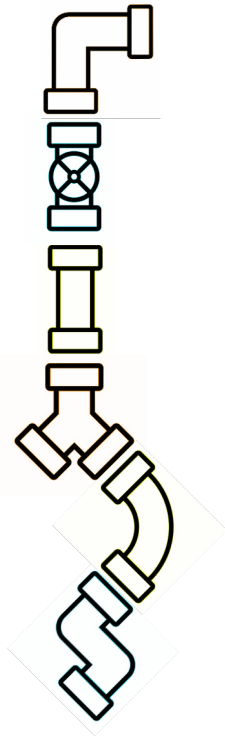
- Examples:
  - Pima Indian Dataset over a period of time (time series data)
  - Input image volume (RGB)

Examples of 4D and 5D tensor data!

# What happens to other tensors' shapes when input shape changes?

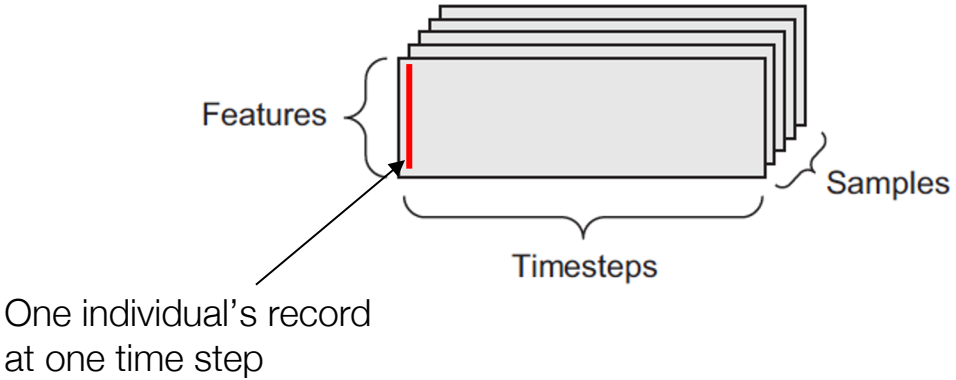
```
model = Sequential()
model.add(Dense(12, input_dim=8, activation='relu'))
model.add(Dense(8, activation='relu'))
model.add(Dense(1, activation='sigmoid'))
print(model.summary())
```



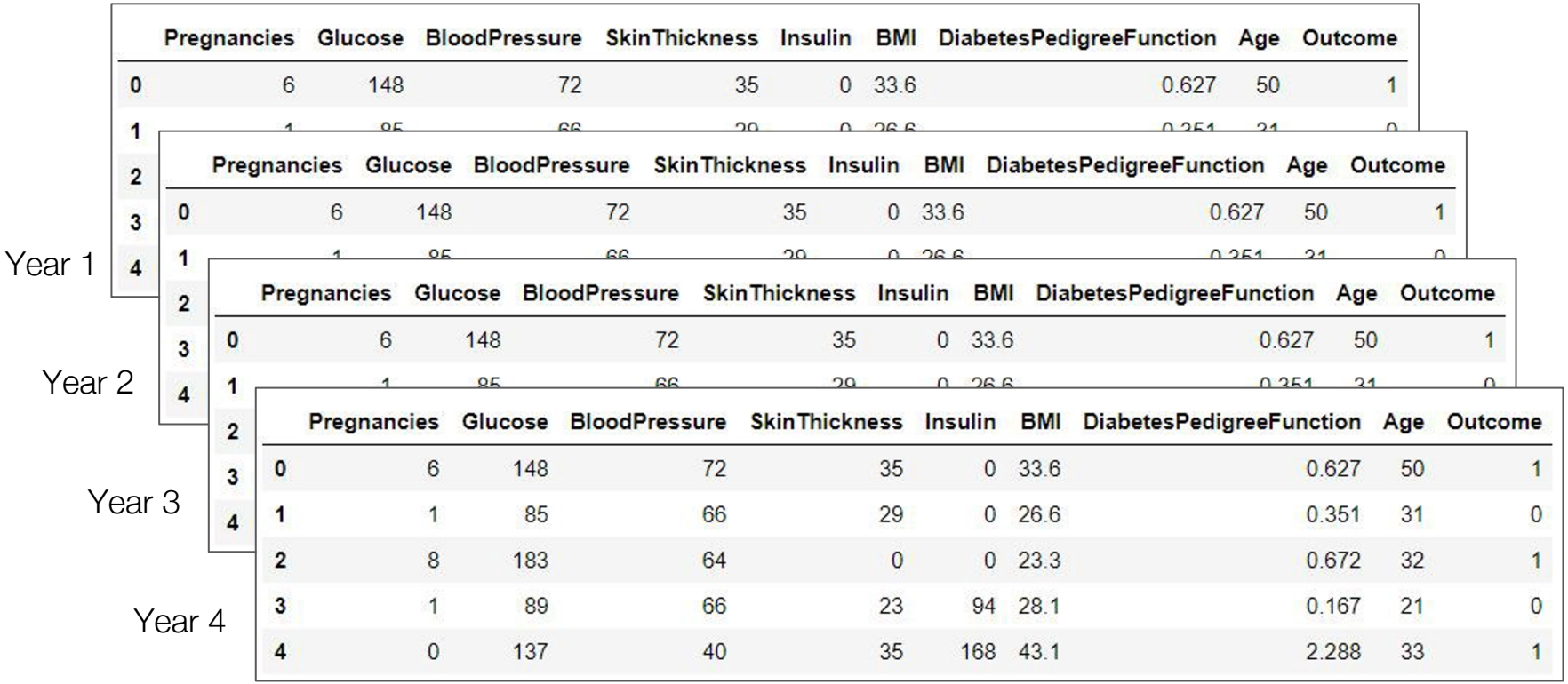



# 2.2.10 Time series data or sequence Data

- 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
- Each sample can be encoded as a sequence of vectors (a 2D tensor), and thus a batch of data will be encoded as a 3D tensor
- The time axis is always the second axis (axis of index 1), by convention



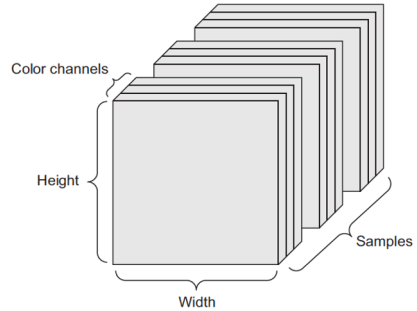
# 2.2.10 Time series data or sequence data




| | Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 6 | 148 | 72 | 35 | 0 | 33.6 | 0.627 | 50 | 1 |
| 1 | 1 | 85 | 66 | 29 | 0 | 26.6 | 0.351 | 31 | 0 |
| 2 | 8 | 183 | 64 | 0 | 0 | 23.3 | 0.672 | 32 | 1 |
| 3 | 1 | 89 | 66 | 23 | 94 | 28.1 | 0.167 | 21 | 0 |
| 4 | 0 | 137 | 40 | 35 | 168 | 43.1 | 2.288 | 33 | 1 |

# 2.2.11 Image data

- Images typically have three dimensions: height, width, and color depth
  - Although grayscale images (like the MNIST digits) have only a single color channel and could thus be stored in 2D tensors, by convention image tensors are always 3D, with a one dimensional color channel for grayscale images
- A batch of 128 grayscale images of size 256 × 256 could thus be stored in a tensor of shape (128, 256, 256, 1), and a batch of 128 color images could be stored in a tensor of shape ?
- There are two conventions for shapes of images tensors: the channels-last convention (used by TensorFlow) and the channels-first convention (Theano).




Is this channels-first or channels-last?

# 2.2.12 Video data

- Video data is one of the few types of real-world data for which we need 5D tensors
- A video is a sequence of frames, each frame being a color image
  - Because each frame can be stored in a 3D tensor (height, width, color_depth), a sequence of frames can be stored in a 4D tensor
    - (frames, height, width, color_depth)
  - Thus a batch of different videos can be stored in a 5D tensor of shape
    - (samples, frames, height, width, color_depth)

- 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, then each value would be stored in 32 bits, so the tensor would represent 405 MB
  - (The data is usually compressed using MPEG format)

# 2.3 The gears of neural networks: Tensor operations

- Building our network by stacking Dense layers on top of each other:

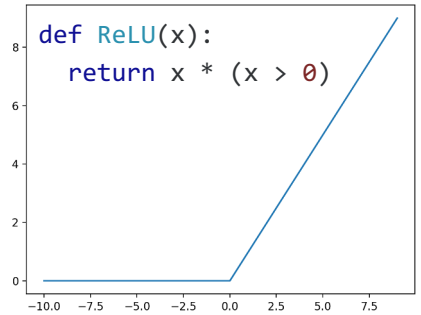
```
keras.layers.Dense(512, activation='relu')
```

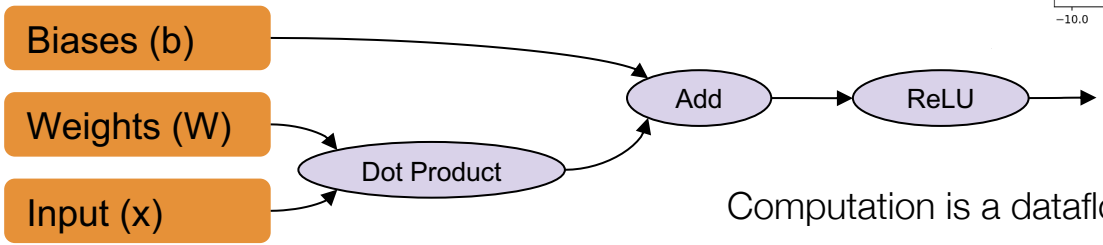
- This layer can be interpreted as a function, which takes as input a 2D tensor and returns another 2D tensor (where W is a 2D tensor and b is a vector)

```
output2D = relu(dot(W, input2D) + b)
```

- We have three tensor operations here:
  - a dot product (dot) between the input tensor and a tensor named W
  - an addition (+) between the resulting 2D tensor and a vector b
  - a relu operation - relu(x) is max(x, 0)

```
def ReLU(x):
  return x * (x > 0)
```




Where is broadcasting?

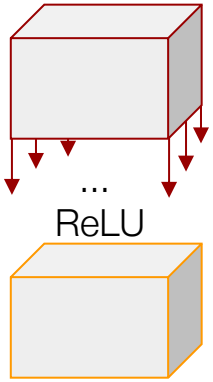
Computation is a dataflow graph (DAG)

# 2.3.1 Element-wise operations

- Naive implementation of an element-wise relu operation:

```
def naive_relu(x):
    assert len(x.shape) == 2    <-- x is a 2D Numpy tensor.

    x = x.copy()                <-- Avoid overwriting the input tensor.
    for i in range(x.shape[0]):
        for j in range(x.shape[1]):
            x[i, j] = max(x[i, j], 0)
    return x
```




- In Numpy, these operations are available as well-optimized built-in Numpy functions
  - These delegate the heavy lifting to a Basic Linear Algebra Subprograms (BLAS) / cuBLAS
  - BLAS are low-level, highly parallel, efficient tensor-manipulation routines implemented in Fortran/C

```
import numpy as np

z = x + y           <-- Element-wise addition

z = np.maximum(z, 0.)           <-- Element-wise relu
```

# 2.3.2 Broadcasting

- With broadcasting, we can apply two-tensor element-wise operations if one tensor has shape (a, b, … n, n + 1, … m) and the other has shape (n, n + 1, … m)

```
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).

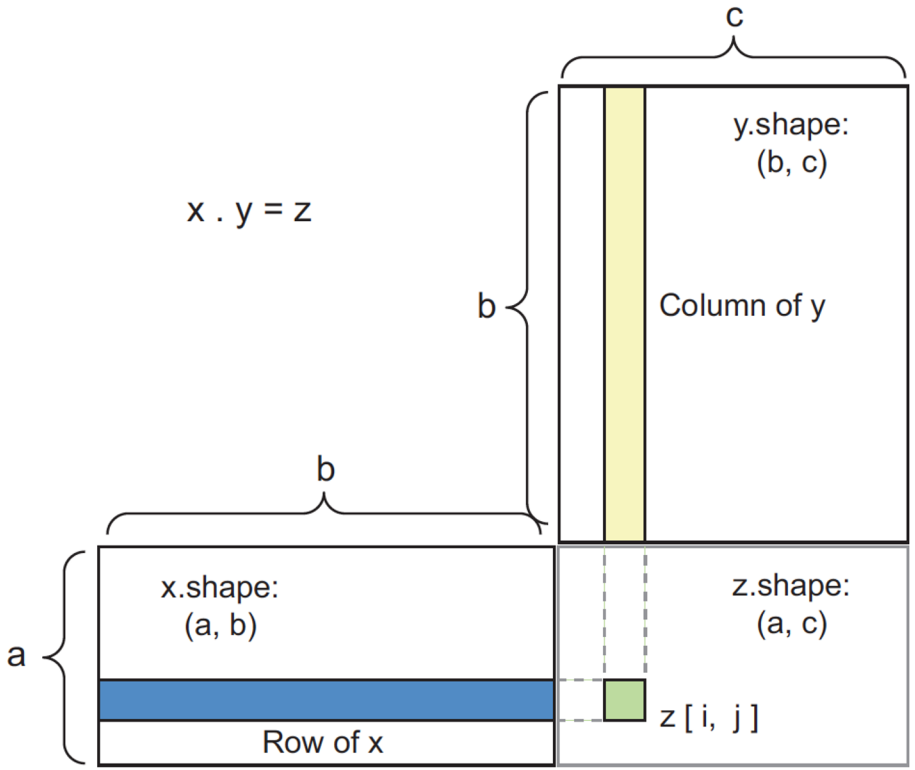
y is a random tensor with shape (32, 10).

The output z has shape (64, 3, 32, 10) like x.

# 2.3.3 Tensor dot (matrix dot product)

```
import numpy as np

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



# 2.3.4 Tensor reshaping

- Reshaping a tensor is rearranging its rows and columns to match a target shape
- The reshaped tensor has the same total number of coefficients as the initial tensor

```
train_images = train_images.reshape(( 60000, 28, 28, 1 ))
```

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

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

Creates an all-zeros matrix of shape (300, 20)

# 2.3.5 Geometric interpretation of tensor operations

- The ‘contents of the tensors manipulated by tensor operations’ can be interpreted as ‘coordinates of points in some geometric space’
  - This implies that all tensor operations have a geometric interpretation
- If `A = [0.5, 1]` is a vector, we can picture the vector as an arrow linking the origin to the point

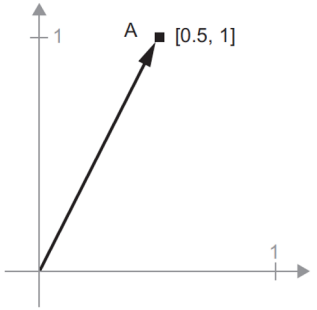



**Figure 2.7 A point in a 2D space pictured as an arrow**

- We add a new point, `B = [1, 0.25]` to A
  - the result is a new location, a vector representing the sum of the previous two vectors

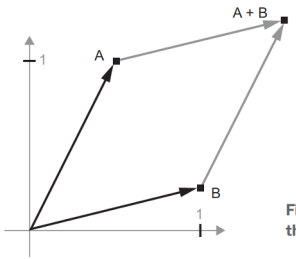
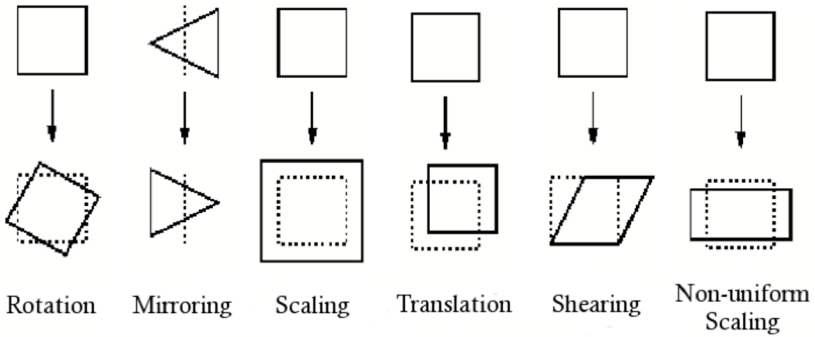



**Figure 2.8 Geometric interpretation of the sum of two vectors**

# 2.3.5 Geometric interpretation of tensor operations

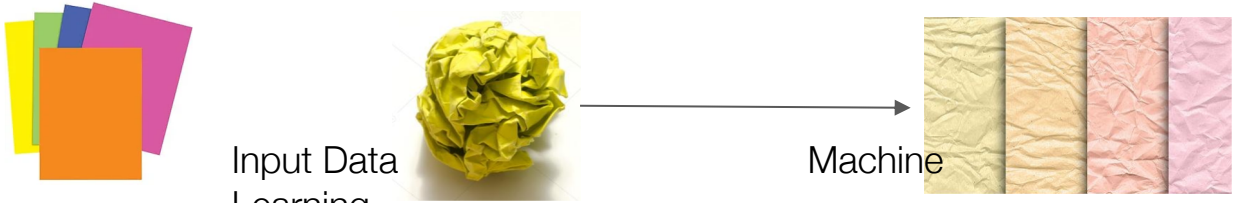
- This means that elementary geometric operations such as affine transformations, (rotations, scaling, etc.) can be expressed as tensor operations




- For instance, a rotation of a 2D vector by an angle `theta` can be achieved via a dot product with a 2 × 2 matrix `R = [u, v]`, where `u` and `v` are both vectors of the plane: `u = [cos(theta), sin(theta)]` and `v = [-sin(theta), cos(theta)]`

# 2.3.6 Geometric interpretation of deep learning

- Neural networks consist entirely of chains of tensor operations and that all of these tensor operations are just geometric transformations of the input data
  - We can interpret a neural network as a very complex geometric transformation in a high-dimensional space, implemented via a long series of simple steps
- Imagine two sheets of colored paper: one red and one blue
  - Put one on top of the other & crumple them together into a small ball
  - The 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 (or any other machine-learning model) 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



# 2.4 The engine of NNs: Gradient-based optimization

- A simple neural network:

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

- Initially, these weight matrices are filled with small random values
  - a step called random initialization
- Training is the gradual adjustment of the weights
- Training loop:
  1. Draw a batch of training samples `x` and corresponding targets `y`
  2. Run the network on `x` to obtain predictions `y_pred` (forward pass)
  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 (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