Deep Learning with Python Ch. 2 Part 2

Tensor Operations

Just as computer programs are reduced to a small set of binary operations (AND, OR, NOT, . . .) the transformations learned by neural networks can be reduced to a handful of *tensor operations* applied to tensors of numeric data.
In the MNIST example the network was built by stacking *Dense* layers on top of each other. One of the *keras* layers looks like:

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

This layer can be interpreted as a function that takes a 2D tensor as input and returns another 2D tensor, a new way of representing the input tensor. Specifically what this is doing is something like:

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

Element-wise operations

- The *relu* (rectified linear unit) operation and addition are element-wise operations that are applied independently to each entry in the tensors being considered. This means that these operations are highly parallelizable.
- A naive Python implementation of *relu* would look like:

```
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
```

Element-wise ops continued

- Similarly addition looks like:

```
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
```

- Of course, in practice there are incredibly efficient Numpy implementations of these operations already and you would just do:

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

Tensor dot

- The dot operation, also called a *tensor product* (not to be confused with an element-wise product) is the most common, most useful tensor operation. Contrary to element-wise operations, it combines entries in the input tensors.
- An element-wise product is done with the * operator in Numpy. The dot operation is done using the dot operator:

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

Tensor dot continued

- So, what does the dot operation do? Let's start with the dot product of two vectors x and y. It is:

```
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
```

So, the result is a scalar and that only vectors with the same number of elements are compatible with dot product.

Tensor dot yet again

- You can also take the dot product between a matrix x and a vector y, which returns a vector where the elements are the dot products between y and the rows of x. 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
```

Notice that dot is no longer symmetric, dot(x,y) isn't the same as dot(y,x).

Matrix tensor dot

- The most common form of tensor dot is probably the 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 elements are the vector products between the rows of x and the columns of y. As follows:

```
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, :]
            col-y  = y[:,j]
            z[i,j] = naive-vector-dot(row-x,col-y)
    return z
```

Tensor reshaping

- Another important tensor operation is *tensor reshaping*. It wasn't used in the *Dense* layers in the MNIST example, but it was used in the pre-processing of the data.

```
train-images =
    train-images.reshape((60000, 28*28))
```

- Reshaping means rearranging its rows and columns to match a target shape. Naturally the reshaped tensor has the same total number of elements as the original. It just adjusts how they are spread around rows and columns.
- One common use for of reshaping is transposing, which just swaps rows and columns of a matrix.

Geometric interpretation

- Because the contents of tensors manipulated by tensor operations can be interpreted as points in a geometric space, all tensor operations have a geometric interpretation. Addition of vectors is a kind of straightforward example.
- So neural networks consist entirely of chains of tensor operations and all of these tensor operations are just geometric transformations of input data. It follows that you can interpret a neural network as a very complex geometric transformation in high dimensional space, implemented as a series of relatively simple steps.
- A 3D mental image may be helpful here. Imagine two sheets of paper, one red and one blue, crumpled up into a ball. That crumpled paper is your input data. Distinguishing the two sheets is a classification problem. So, it analogizes to the steps required to uncrumple the ball and separate the two pieces of paper.