Solution to Linear Node

Here's my solution to the last quiz:

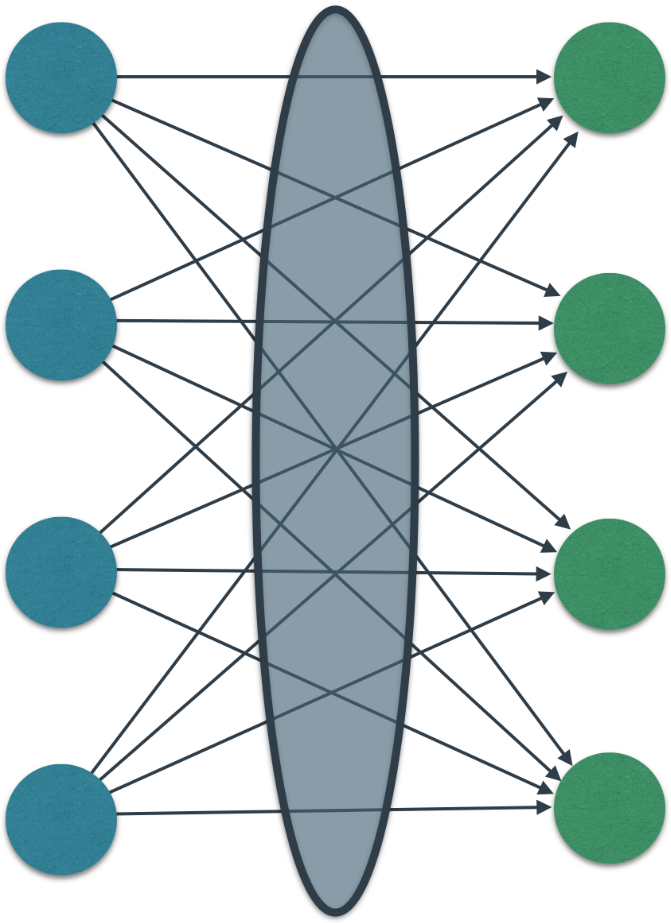
```
class Linear(Node):
    def __init__(self, inputs, weights, bias):
        Node.__init__(self, [inputs, weights, bias])

    def forward(self):
        """
        Set self.value to the value of the linear function output.

        Your code goes here!
        """
        inputs = self.inbound_nodes[0].value
        weights = self.inbound_nodes[1].value
        bias = self.inbound_nodes[2].value
        self.value = bias
        for x, w in zip(inputs, weights):
            self.value += x * w
```

In the solution, I set `self.value` to the bias and then loop through the inputs and weights, adding each weighted input to `self.value`. Notice calling `.value` on `self.inbound_nodes[0]` or `self.inbound_nodes[1]` gives us a list.

Transform



Shift your thinking here to the edges between layers.

Linear algebra nicely reflects the idea of transforming values between layers in a graph. In fact, the concept of a transform does exactly what a layer should do - it converts inputs to outputs in many dimensions.

Let's go back to our equation for the output.

$$o = \sum_i x_i w_i + b$$

Equation (1)

For the rest of this section we'll denote *x* as *X* and *w* as *W* since they are now matrices, and *b* is now a vector instead of a scalar.

Consider a `Linear` node with 1 input and k outputs (mapping 1 input to k outputs). In this context an input/output is synonymous with a feature.

In this case X is a 1 by 1 matrix.

$$X = \begin{bmatrix} X_{11} \end{bmatrix}$$

1 by 1 matrix, 1 element.

W becomes a 1 by k matrix (looks like a row).

$$W = \begin{bmatrix} W_{11} & W_{12} & W_{13} & \dots & W_{1k} \end{bmatrix}$$

A 1 by k weights row matrix.

The result of the matrix multiplication of X and W is a 1 by k matrix. Since b is also a 1 by k row matrix (1 bias per output), b is added to the output of the X and W matrix multiplication.

What if we are mapping n inputs to k outputs?

Then X is now a 1 by n matrix and W is a n by k matrix. The result of the matrix multiplication is still a 1 by k matrix so the use of the biases remain the same.

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} & \dots & X_{1n} \end{bmatrix}$$

X is now a 1 by n matrix, n inputs/features.

$$W = \begin{bmatrix} W_{11} & W_{12} & W_{13} & \dots & W_{1k} \\ W_{21} & W_{22} & W_{23} & \dots & W_{2k} \\ W_{31} & W_{32} & W_{33} & \dots & W_{3k} \\ \vdots & & & \ddots & \vdots \\ W_{n1} & W_{n2} & W_{n3} & \dots & W_{nk} \end{bmatrix}$$

W is now a n by k matrix.

$$b = \begin{pmatrix} b_1 & b_2 & b_3 & \dots & b_k \end{pmatrix}$$

Row matrix of biases, one for each output.

Let's take a look at an example of n inputs. Consider an 28px by 28px greyscale image, as is in the case of images in the MNIST dataset. We can reshape the image such that it's a 1 by 784 matrix, n = 784. Each pixel is an input/feature. Here's an animated example emphasizing a pixel is a feature.

In practice, it's common to feed in multiple data examples in each forward pass rather than just 1. The reasoning for this is the examples can be processed in parallel, resulting in big performance gains. The number of examples is called the *batch size*. Common numbers for the batch size are 32, 64, 128, 256, 512. Generally, it's the most we can comfortably fit in memory.

What does this mean for *X*, *W* and *b*?

X becomes a m by n matrix and *W* and *b* remain the same. The result of the matrix multiplication is now m by k, so the addition of *b* is **broadcast**over each row.

$$X = \begin{bmatrix} X_{11} & X_{12} & X_{13} & \dots & X_{1n} \\ X_{21} & X_{22} & X_{23} & \dots & X_{2n} \\ X_{31} & X_{32} & X_{33} & \dots & X_{3n} \\ \vdots & & & \ddots & \vdots \\ X_{m1} & X_{m2} & X_{m3} & \dots & X_{mn} \end{bmatrix}$$

X is now an m by n matrix. Each row has n inputs/features.

In the context of MNIST each row of *X* is an image reshaped from 28 by 28 to 1 by 784.

Equation (1) turns into:

$$Z = XW + b$$

Equation (2).

Equation (2)

Equation (2) can also be viewed as $Z = XW + B$ where B is the biases vector, b, stacked m times as a row. Due to broadcasting it's abbreviated to $Z = XW + b$.

I want you to rebuild `Linear` to handle matrices and vectors using the venerable Python math package `numpy` to make your life easier. `numpy`is often abbreviated as `np`, so we'll refer to it as `np` when referring to code.

I used `np.array` (documentation) to create the matrices and vectors. You'll want to use `np.dot`, which functions as matrix multiplication for 2D arrays (documentation), to multiply the input and weights matrices from Equation (2). It's also worth noting that numpy actually overloads the `__add__` operator so you can use it directly with `np.array` (eg. `np.array() + np.array()`).

Instructions

1. Open nn.py. See how the neural network implements the `Linear`node.
2. Open miniflow.py. Implement Equation (2) within the forward pass for the `Linear` node.
3. Test your work!