First look at CeTZ

Introduction

One of the most important features of a typesetting language is the ability to create diagrams and figures programmatically (for instance to represent neural network architectures or other schematics). In LaTeX, this is done with TikZ (there are other ways but I'm more used to TikZ) but TikZ is not available in Typst. Instead, we need to use another package called CeTZ (the acronym for *CeTZ: ein Typst Zeichenpaket*).

Getting started

In this document, I start exploring the CeTZ package using the getting started guide. However I plan to create different diagrams and I won't be following this guide 100%, it's more of a reference to learn the package.

To start using CeTZ, we need to import it by adding the following line to the document.

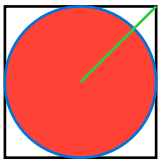
```
#import "@preview/cetz:0.3.1"
```

This line has already been added to the beginning of this document and imports the version 0.3.1 of the CeTZ package.

To add CeTZ *canvas*, where CeTZ content can be drawn, we need to use the `cetz.canvas` function, passing a code block that returns all content that has to be drawn. Drawing functions are located in the `cetz.draw` module that thus needs to be imported before starting using CeTZ. It could be imported at the beginning of the document but since it overrides some Typst core functions (such as `line` or `circle`), it's cleaner to import it in each canva block. So to start drawing stuff with CeTZ (assuming it has been imported before), we need to write the following.

```
#cetz.canvas({
  import cetz.draw: * // import all objects from cetz.draw

  // start drawing stuff using cetz functions
  ...
})
```

For instance, here is a figure taken for CeTZ documentation with the accompagning code.



```
#cetz.canvas({
    import cetz.draw: *
    rect((-1, -1), (1, 1))
    set-style(stroke: blue, fill: red)
    circle((0,0))
    line((), (1,1), stroke: green)
})
```

FNN diagram

What looks like a good first figure that I'm excited to try is drawing a Feedforward Neural Network (FNN), one of the simplest neural network architecture. This kind of neural networks is made out of layers, each layer containing neurons (nodes) that are connected to all neurons from the previous and the next layers.

A FNN can have any number of layers (this parameter is called the *depth*) and each layer can have any number of neurons (this other parameter is the *width*).

I have my own LaTeX macro for drawing FNN (see here), but only the width is variable. Implementing a variable depth would require the `xargs` package and I prefered avoiding it for this use case to avoid adding unecessary complexity. However, the way Typst and CeTZ work look like the perfect occasion to implement a `fnn` function that could draw a neural network of any width and any depth and that's what I'll be implementing here.

Specification

For now, let's start simple. Our `fnn` function will only take a single argument `layers`, which will be an array of integers. In this array, the i^{th} element is the width of layer i (the number of neurons in this layer). The depth of the neural network is simply `layers.len()`.

As a result, the `fnn` function should draw the corresponding Feedforward Neural Network, each neuron being properly labeled just like in my TikZ implementation.

Implementation

This is how the `fnn` function is implemented (I've also defined it in the source code in addition to the following code block).

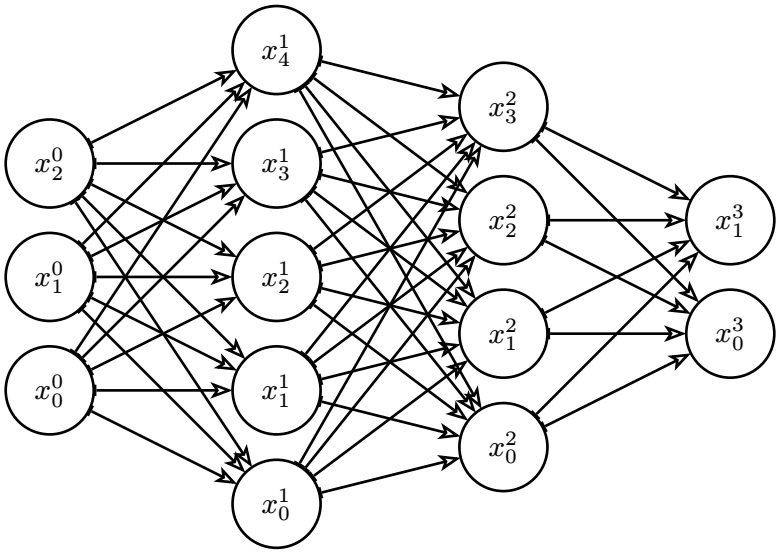
```
#let fnn(layers) = [
  #cetz.canvas({
    import cetz.draw: *
    let x_offset = 3cm // horizontal spacing between layers
    let y_offset = 1.5cm // vertical spacing between neurons

    // set mark style for connection between neurons
    // start mark is set to bar so it can be offset by the pos argument because pos
cannot be applied if there is no mark. Idealy an invisible mark could be added as an
easy solution?
    set-style(mark: (start: "bar", end: "stealth", pos: 1.5em))

    // function that returns the position of a neuron given its layer and its number
    let neuron_pos(layer_num, neuron) = {
      let spatial_width = y_offset * layers.at(layer_num) // layer width in cm
(actually height since layers are vertical)
      (layer_num*x_offset, {neuron*y_offset - spatial_width/2})
    }
    for layer_num in range(layers.len()) {
      let layer = layers.at(layer_num)
      for neuron in range(layer) {
        let coords = neuron_pos(layer_num, neuron)
        // weirdly enough, CeTZ can't directly put content inside circle. It needs to
be handled by a separate `content` function.
        circle(coords, radius: 1.5em)
        content(coords, anchor: "base")[$x_#neuron^#layer_num$]

        // finally, connect this neuron to all neurons of previous layer (if not in
the first layer)
        if 0 < layer_num {
          for other in range(layers.at(layer_num - 1)) {
            let other_coords = neuron_pos(layer_num - 1, other)
            line(other_coords, coords)
          }
        }
      }
      let layer = layer + 1
    }
  })
]
```

Usage

Once the `fnn` function is defined, we can call it the following way.

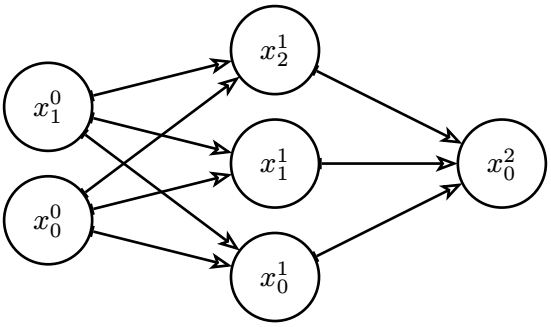
```
#fnn((3, 5, 4, 2))
```

Which looks like this.



Note that I used the `align` command to horizontally center the diagram.

We can draw small networks:



Or bigger ones: