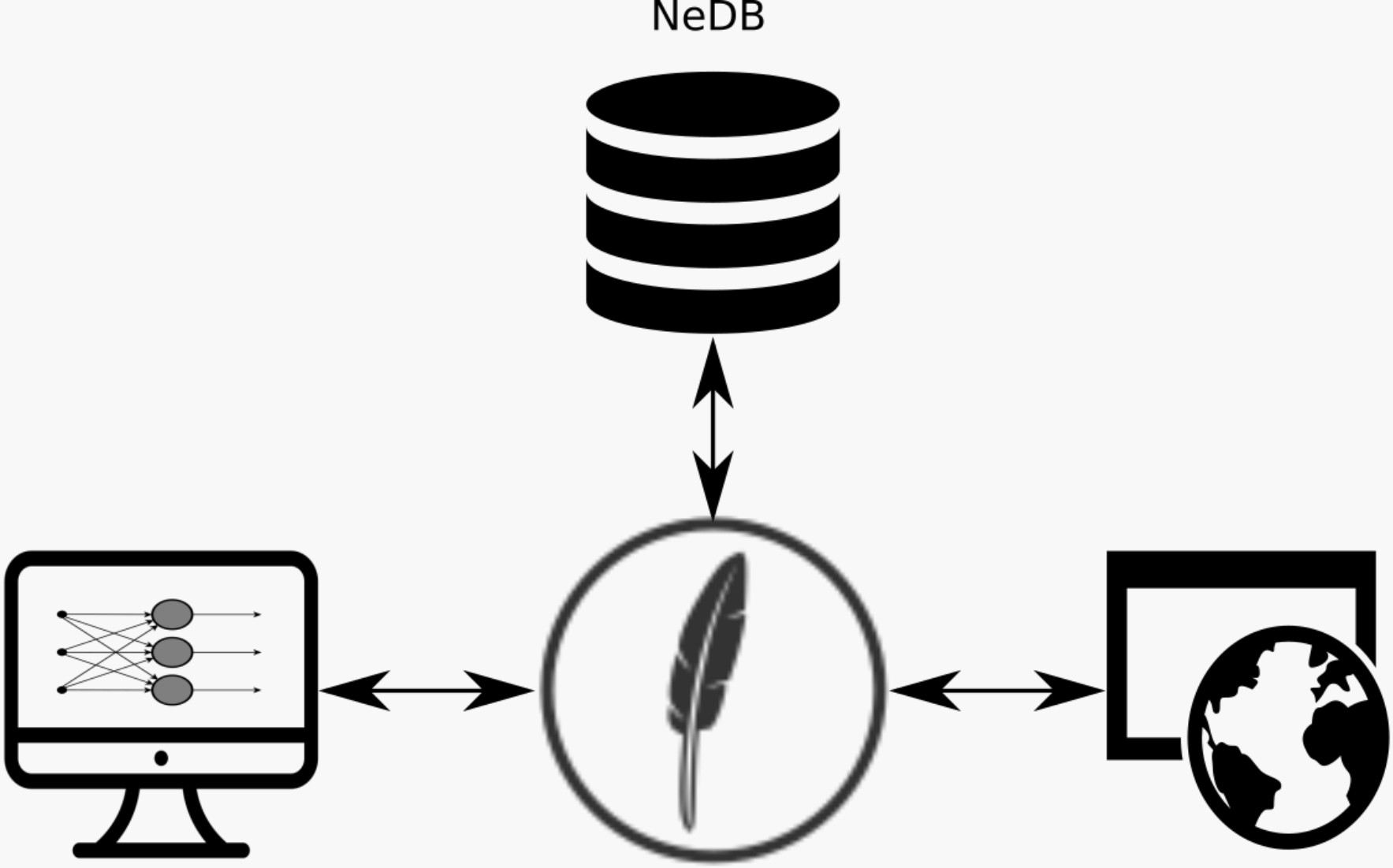
Agnez, analytics for deep learning research

WRITTEN BY EDER SANTANA

Machine learning is about writing programs with parameters that are learned from data. But writing the base architecture that will be learned requires intuition, inspection, trial, and error, all elements that can be enhanced with high quality visualization and analytics tools.

Building visualization and analytics tools to assist deep learning (and machine learning in general) development was what motivated my brother Tiago Santana and I to start Agnez, a collection of visualization tools for deep learning. Models used for deep learning can be seen a business where the architecture and hyperparameters are the business choices and the accuracy or error in the test set a measure of business success. Keeping that metaphor in mind we looked for companies such as Keen IO and projects such as the Automatic Statistician for inspiration to build research analytics and visualization tools.

Here we will describe our approach to serve the visualizations as a web app using Feathers.js in the backend and Keen Dashboards in the frontend. For generating the graphs we are using a temporary solution based on mpld3 that converts Matplotlib graphs to D3. The full code is on minimal-app repository. A schematic diagram of our architecture is shown in the figure below.



We wanted to generate beautifully organized dashboards and we noticed that Keen Dashboards already lifted most of the design weight. But as an originally Python developer, I suggested to keep Matplotlib's subplot arrangement and flexibility without needing to rewrite html ourselves. An elegant solution to this problem would be to generate the dashboards dynamically using a REST API. We chose to develop the API with Feathers.js, a thin wrapper around Express.js for building real time REST APIs with Node.js. This is what we needed to start a simple to use and general API for handling, storing and plotting model analytics. In coffeescript and using NeDB as the database, our Feathers app is simply:

```
feathers = require 'feathers'
mongodb = require('feathers-mongodb')
memory = mongodb {
  db: 'edermempy'
  collection: 'values'
}

bodyParser = require 'body-parser'
```

```
app = feathers()

app.configure feathers.rest()
  .configure feathers.socketio()
  .use bodyParser.json()
  .use '/values', memory
  .use '/', feathers.static(__dirname)
  .listen 3000

console.log 'App listening on port 3000
console.log 'Index at', __dirname+'/static/'
```

All the hard work is handled by feathers-nedb CRUD and feathers-client that uses socket.io to update the browser client in real time. The machine learning client training our model with Python sends POST requests to the server. These requests trigger events in the server that updates the browser page. For this simple demo, our Python client will send html strings generated with mpld3 and a gif. When training a deep learning model the html string would be graphs of cost functions, accuracy, weight norms and other useful analytics. As we mentioned this is a simple temporary solution for illustration purposes, it would be more general to use a native D3 chart, patch the graphs in the browser side and only send numbers from the machine learning side. Nevertheless, deep learning epochs, or passes through the training datasets, usually take a few seconds (or even minutes and hours depending on how large the training dataset is) and sending html strings does not add a considerable overhead.

The index.html is pretty minimal since everything will be generated dynamically when we send data using the API. We start with a simple `<div id=dashboard>` and add new Bootstrap rows later. `script.coffee` in the server has a basic Keen Dashboard cell as follows:

```
String.prototype.format = ->
  args = arguments
  return this.replace /{(\d+)}/g, (match, number) ->
    return if typeof args[number] isnt 'undefined' then args[number]

cellstr = """
  <div class="col-sm-6">
    <div class="chart-wrapper">
      <div class="chart-title" id=title{1}>
        {0}
      </div>
      <div class="chart-stage" id="grid{1}">
        {2}
      </div>
      <div class="chart-notes" id="description{1}">
```

```
          {3}
        </div>
      </div>
    </div>
"""
```

With this code we can fill the placeholders using a Python inspired syntax: `"dat {0} is {1}".format "string", "cool"` which returns `"dat string is cool"`. When creating a new cell, we simply append the filled string to `#dashboard`'s html. When the html string or an image URL is patched, we update that dashboard cell using the snippet below:

```
values.on 'patched', (val) ->
  console.log 'patching', val.name
  $grid = $ "#grid#{val.pos}"
  $title = $ "#title#{val.pos}"
  $description = $ "#description#{val.pos}"

  $title.html val.name
  $description.html val.description
  if val.type is "html"
    $grid.html val.value
  if val.type is "img"
    $grid.html "<img src='#{val.value}'>"
```

To test the app, we use the Python script ahead.

```
# Remember that we are using feathers database CRUD

# Allocate cell space in the dashboard by calling the CREATE method
url = "./images/main_img.gif"
r = requests.post("http://localhost:3000/values",
                  json={'name': '', 'type': 'html', 'value': [], 'pos
                        'description': ''})
id0 = json.loads(r.text)["_id"]
r = requests.post("http://localhost:3000/values",
                  json={'name': '', 'type': 'img', 'value': [], 'pos'
                        'description': ''})
id1 = json.loads(r.text)["_id"]
fig = plt.figure()
numbers = []

# Update the cell html calling the PATCH method
for i in range(100):
    time.sleep(2) # simulate wait time of an epoch
    plt.clf()
    numbers.append(random.random()) # new value
    plt.plot(numbers)
    if len(numbers) > 20:
        del numbers[0] # delete old values
```

```
    html = mpld3.fig_to_html(fig) # convert matplotlib to d3
    # PATCH requests
    r = requests.patch("http://localhost:3000/values/" + str(id0),
                       json={'name': 'test1', 'type': 'html', 'value'
                             'pos': 0, 'description': 'simple test'})
    r = requests.patch("http://localhost:3000/values/" + str(id1),
                       json={'name': 'test2', 'type': 'img', 'value':
                             'pos': 1, 'description': 'simple image'}
    print r
```

Note that since REST APIs are universal, we could send pictures and graphs with any other language. In the next iteration of this app, using native D3 charts generated in the browser side we will make even easier to serve visualizations in a way that is language agnostic to the machine learning side (Lua and C++ are also popular for deep learning).

For those interested in playing with this code, from the source root directory run

```
coffee app.coffee
```

to start up the app and run

```
python test.py
```

to send data using the API. We can see the results at http://localhost:3000 and https://localhost:3000/values

If you are training a deep learning model with Keras you can run the app and use the Keras callbacks we provide, as in this example.

Since Agnez is an young project, we are expecting it evolve quickly. Help, suggestions and feedback are welcome.

« Full blog

These are personal words. Here you may find references to unconventional ideas. Please, read it at your own risk or don't read it at all. If you are looking for my conventional science publications, please visit this website.