Exercise 10.1: Playing with a neuronal network vizualization

Open the website `https://playground.tensorflow.org/`, which visualizes the training of a neuronal network. In this case, the network is trained to learn a function $f(x, y)$ which is specified only by a set of data points (being blue or orange) at various x, y positions, plotted as the dots in the output graph. Vary the number of layers, the number of neurons per layer and the input data set and observe how this affects the training. What is the best layout if you fix the number of neurons?

Exercise 10.2: Neuronal Network training

Download the python files[1] `network.py`, `load_data.py` and the MNIST data set `mnist.pkl.gz` provided on the course website.

a) The MNIST data set contains (a lot of) images of hand-written digits, along with labels 0-9 of the shown number. Load the data set with the function `load_data_wrapper` of `load_data.py` and plot a few of the images.

   Hint: Throughout this exercise, we don't need the `validataion_data` returned by the function. The `test_data` is a list of tuples `(image, label)`. To plot an image, reshape it to 28×28 pixels and use the `imgshow` function of Matplotlib.

b) Read the code of `network.py`.

c) The `Network` class of `network.py` lacks an implementation of the backpropagation algorithm in `backprop(self, x, y)`. Implement it!

   Note: You can find working code in the online book by Nielsen. However, the point of this exercise is to try the implementation yourself!

   Hints: The goal is to calculate the gradient of the cost function $C = \frac{1}{2}\|a^L - y\|^2$ with respect to each entry of the weights w^l (matrices) and biases b^l (vectors) in each layer l. Here, x (vector) is the input to image for the network, y (vector) the desired output, and a^L (vector) is the activation of the neurons in the last layer L for the given input x. In this code, vectors (including the input images) are numpy arrays shaped as column vectors $(len, 1)$.

   - Begin with a code similar as in `Network.feedforward` to calculate the weighted input z^l and activations a^l for each layer $l = 1, \ldots, L$, using the definitions

     $$a^l = \sigma(z^l), \qquad\qquad z^l = w^l a^{l-1} + b^l \tag{1}$$

     starting with $a^0 = x$. Here, $\sigma(z)$ labels the sigmoid function as defined in `network.py`.

[1] The provided code is based on (code from) the (free) online-book *Neural networks and deep learning* by M. Nielsen, `http://neuralnetworksanddeeplearning.com`.

- Calculate
$$\delta^L \equiv \frac{\partial C}{\partial z^L} = \frac{\partial \frac{1}{2}\|\sigma(z^L) - y\|^2}{\partial z^L} = (a^L - y) * \sigma'(z^L), \tag{2}$$
  where $*$ labels the element-wise product (as performed by the `*` between two numpy arrays) and $\sigma'(z)$ is the derivative of the sigmoid function. This is the only point which depends on the exact form of the used cost-function C. For this reason, there is a (trivial) function `self.cost_derivative` defined, which you can use to calculate $(a^L - y)$.

Using the definitions eq. (1), the chain rule implies for $l = L-1, \dots, 1$:
$$\delta^l \equiv \frac{\partial C}{\partial z^l} = \underbrace{\frac{\partial C}{\partial z^{l+1}}}_{\delta^{l+1}} \underbrace{\frac{\partial z^{l+1}}{\partial a^l}}_{(w^{l+1})^T} \underbrace{\frac{\partial a^l}{\partial z^l}}_{\sigma'(z^l)} = \left((w^{l+1})^T \delta^{l+1}\right) * \sigma'(z^l). \tag{3}$$

Here, $(w^{l+1})^T$ denotes the usual matrix transpose, the need of which becomes clear when considering elements of the above vector equation. Moreover, we have:
$$\frac{\partial C}{\partial b^l} = \underbrace{\frac{\partial C}{\partial z^l}}_{\delta^l} \underbrace{\frac{\partial z^l}{\partial b^l}}_{1} = \delta^l \tag{4}$$
$$\frac{\partial C}{\partial w^l} = \underbrace{\frac{\partial C}{\partial z^l}}_{\delta^l} \underbrace{\frac{\partial z^l}{\partial w^l}}_{(a^{l-1})^T} = \delta^l \times (a^{l-1})^T \tag{5}$$

The last product of eq. (5) is to be read as a matrix-product column-vector $\times$ row-vector to yield a matrix.

- Use eq. (3), to back-propagate the δ^l from the last layer $l = L$ to the first (hidden) layer $l = 1$ and calculate the gradients using eqs. (4),(5).

d) Train a network `Network([784, 30, 10])` with a single hidden layer. What accuaracy do you achieve?

   Hint: If you implemented the backpropagation correctly, you should get accuracies of up to $\approx 94\%$!

e) Train other networks with more hidden layers and/or more neurons and try to increase the final accuracy.

So far we have trained a neuronal network to classify images of digits. Can we use such a setup in physics? One idea[2] is to train a network to classify different phases. The provided file `generate_mc_data.py` runs a Monte-Carlo simulation of the classical Ising model (using the code we wrote in earlier exercises) and saves snapshots of the spin configurations during the simulation at 3 different temperaturs $T = 1.5, 2.3, 3$.

f) Run the file `generate_mc_data.py` to generate the data file `mcIsing.pkl.gz` (which may take a few minutes). This file uses the same format as the MNIST data set (except that is uses only 3 different labels).

[2] See for example the paper by J. Carrasquilla and R. Melko, https://arxiv.org/abs/1605.01735.

g) Load the data (with the same function as for the MNIST data set) and plot a few of the images. Can you assign the temperatures to the labels?

h) Train a neural network with a single layer of 30 hidden neurons to classify the digits. What accuracy can you achieve? Try again with a deeper network (i.e., more layers) using the same number of neurons as before. What accuracy can you achieve now? Would a perfect neural network be able to classify 100% of the snapshots correctly?

   Hint: You need to adjust the number of output neurons to the number of temperature values, here 3.