:Assignment 3

MNIST Digits Classification using Neural Networks

Due date: See in Moodle

:The problem

Our problem deals with digits classification. We will train a simple neural network that receives an input image of a hand-written digit (0-9), and outputs the class (digit .number) obtained by the trained model

The difficulty of visual pattern recognition is apparent if you attempt to write a computer program to recognize hand-written digits. What seems easy when we do it ourselves suddenly becomes extremely difficult. Simple intuitions about how we recognize shapes - "a 9 has a loop at the top, and a vertical stroke in the bottom right" - turn out to be not so simple to express algorithmically. When you try to make such rules precise, you quickly get lost in a mess of exceptions and special cases. It .seems hopeless

Neural networks approach the problem in a different way. The idea is to take a large number of handwritten digits, known as training examples, and then develop a system which can learn from those training examples. In other words, the neural network uses the examples to automatically infer rules for recognizing handwritten .digits

In the attached iPython notebook **'CV_mnist_net.ipynb'** you will develop and train a .NN for handwritten digits recognition. This will be done using the MNIST database .Just follow the instructions given in the notebook and fill in the missing code .Here you will find helpful theory for basic NN model and training

Setup Instructions

.We highly recommend using the free Google Colaboratory platform

Tutorial for Google Colab:
https://medium.com/deep-learning-turkey/google-colab-free-gpu-tutorial-e113627b9f5d

Another useful tutorial:
https://medium.com/@oribarel/getting-the-most-out-of-your-google-colab-2b0585f82403

To open the file with Colab: save the notebook on your google drive right click on the file open with Google Colaboratory

:The Dataset

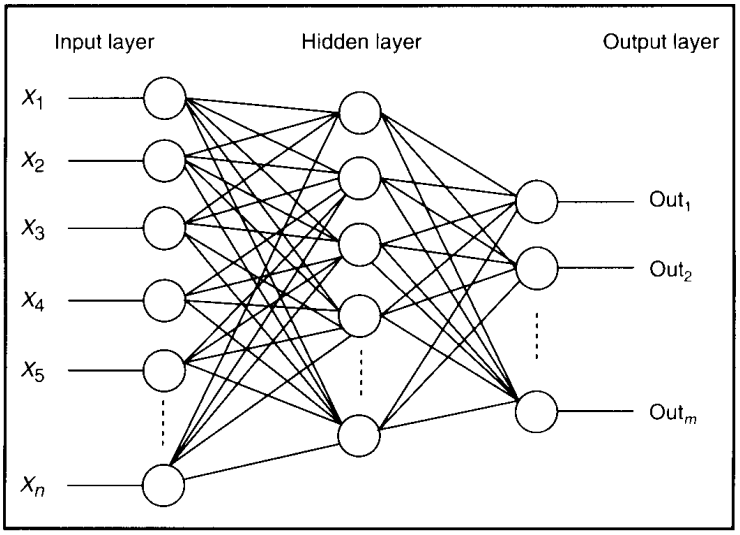
The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples. The digits have been size-normalized and centered to a .(fixed-size image of 28x28 pixels (gray scale

The data will be downloaded in the first cell of the code to the local directory .((containing the notebook

Neural Networks

:The neural network structure

.In this exercise we will implement a two-layer neural network with one hidden layer



The first layer is the input layer, which receives a flattened image of 28x28 (pixels) = **784** .**input cells** in our case

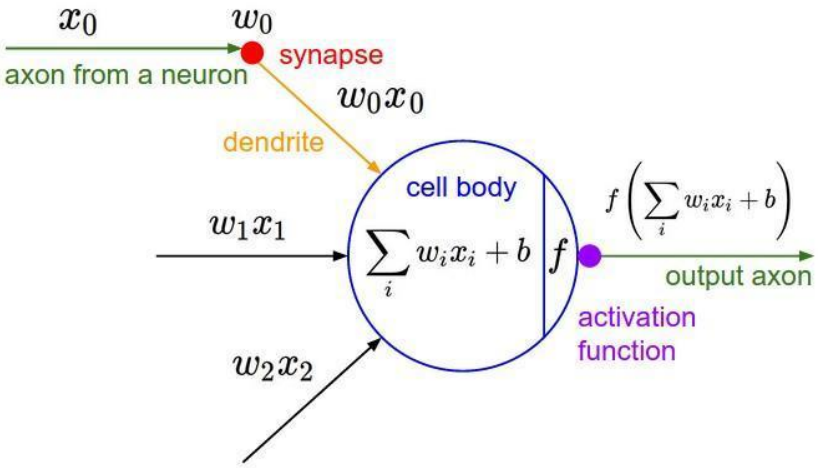
Notice that when we say N-layer neural network, we do not count the input layer .((because it is not trainable

The input layer is linked to a "hidden layer" with full connectivity ("fully connected layer"). The hidden layer contains hidden units (or cells), **the amount of hidden units .is configurable**

The last layer is the output layer. Because we want to classify the output image into a digit between 0 and 9, the output layer will include **10 output cells**. Each output .neuron corresponds to a certain class (digit). The output layer is also fully connected

:Neuron structure

Each cell in the network (called a **neuron**) receives a number of weighed inputs, sums them up and passes them through a nonlinear function (such as sigmoid, hyperbolic tangent, RELU, etc.), which is called "activation function". In addition, one of the .(inputs to each cell is the **bias** (which is not dependent on the previous layers



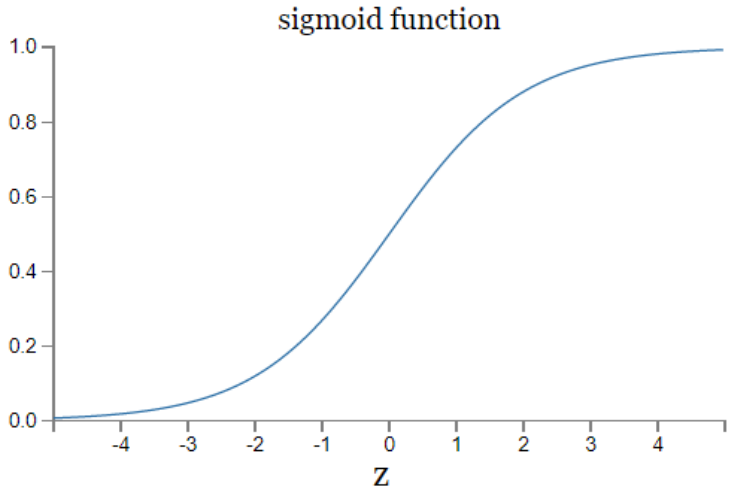
:In this exercise we will be using the **Sigmoid function** for the activation function

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

:Where z is the output of the fully connected layer

$$z = w \cdot x + b$$

Where *x* is the output from the previous layer (or the input of the first layer), *b* is the .bias



.This shape resembles a smoothed-out version of a step function

It takes a real-valued number and “squashes” it into range between 0 and 1. In .particular, large negative numbers become 0 and large positive numbers become 1

:(The Sigmoid's derivative (you may confirm it yourselves

$$\sigma'(z) = \sigma(z)\cdot(1 - \sigma(z))$$

Description of the algorithm - Supervised learning

In the learning process, the weights and bias inputs are updated to map the input image to the correct output cell corresponding to the class represented by the image ((the digit in the image.

:The process contains three main parts that are **repeated**

:Forward Propagation

The information from the input layer passes through the network to the cells in the hidden layer, and the output of the hidden cells continues to propagate to the output layer (the network's output is our "hypothesis", it is a vector marked by h).

Next we compare the network's output with the Ground Truth (GT) digit represented in the image, computing the networks error with the loss function.

We will be using Softmax classifier on our output layer, it takes a vector of arbitrary real-valued scores (in *z*) and squashes it to a vector of values between zero and one that sum to one (similar to probabilities).

:For each output cell, the softmax function

$$h_i = \frac{e^{z_i}}{\sum_k^C e^{z_k}}$$

Cross Entropy Loss: Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is. It is defined as:

$$cost(t,h) = -\sum_i^C t_i log(h_i) = -\ log(h_j)$$

.Where *j* is the index of the correct class

.Cross entropy measure is a widely used alternative of squared error

:Back Propagation

For the optimization purpose, we will want to change the neural network's parameters so that we minimize the error it produces, in other words, we will want the output class to be as close as possible to the GT class. Back propagation allows the information to go back from the cost backward through the network in order to compute the gradients needed for the optimization. We will use Gradient Descent

(or, more precisely, **Stochastic Gradient Descent** because we update the network as .we enter the data – for each 'minibatch') to update the NN's weights

Therefore, first (back propping from the end) - we will derive the error as a function of the network's output, where *t* is a 'one-hot' vector with t(GT digit) = 1 and .t(elsewhere)=0

$$Delta = \frac{\partial Cost}{\partial z} = h - t$$

.Continue to 'back-prop' by yourselves, using the ‘chain rule’ for derivatives

:Example of the chain rule

$$\frac{\partial f}{\partial s} = \frac{\partial f}{\partial r} * \frac{\partial r}{\partial s}$$

.$r(s)$and $f(r)$Where

The purpose of the backpropagation is to find the gradient of the cost function with :(respect to the networks parameters (weights and biases

$$\frac{\partial Cost}{\partial b}$$

$$\frac{\partial Cost}{\partial w}$$

:Update

Now that we have calculated the derivatives, we can update the weights so that they :minimize the error with the gradient descent algorithm

$$w = w - \mu\frac{\partial Cost}{\partial w}$$

.is the learning rate μ Where

:Similarly for the bias

$$b = b - \mu\frac{\partial Cost}{\partial b}$$

This process is done for each 'minibatch' of the training data and the process is done .iteratively until the network converges

:Submitting your assignment

.(The homework will be done in pairs (only one should submit it

Please submit: 1. A **PDF** of the completed iPython

.The **ipynb** file as well .2

The name format for both files should be: **Assignment_2_id-number1_id-number2**

Important: *Please make sure that the submitted notebook has been run and the cells .(outputs are visible (in the PDF and in the notebook*