Machine Learning Programming

Assignment 6: Neural Networks (NN)

This series of practicals focuses on the development of machine learning algorithms introduced in the Applied Machine Learning course. In this practical, you will implement the basis functions to develop a Neural Network algorithm.

Deadline: December 24th, 2021 @ 12pm (12h00)

Assignments must be turned in by the deadline. **1/6 point will be removed for each day late (a day late starts one hour after the deadline).**

Procedure: From the course Moodle webpage, you must download and extract the `.zip` file named `TP6-NN-Assignment.zip` . The folder contains the following elements:

- gui.m
- gui.fig
- **functions**
- **data**
- **utils**

Only the elements in blue need to be filled out. The rest are helper functions or data to be imported. This assignment will be less guided than the previous ones. **There are no evaluation functions provided.**

You must submit an archive as a `.zip` file with the following naming convention `TP6-NN-SCIPER.zip` , where `SCIPER` need to be replaced by your own **sciper number**. You must submit **only** the files/folders in blue. **DO NOT INCLUDE** the folders **data**, and **utils**.

We take plagiarism very seriously. Submitting works that are not yours will result in a report submitted to the competent EPFL authorities.

Part 1: Binary classification with Neural Networks

For this assignment, an implementation of a Neural Network is provided as a class `NeuralNetwork.m` in the `utils` folder. The class implements all the methods seen during the lecture of applied machine learning. The next sections will cover the different elements already implemented in the provided code.

As a reminder, the core principle of Neural Networks is, given an input $\mathbf{X} \in \mathbb{R}^N$ to define a mapping $f : \mathbf{X} \rightarrow \mathbf{Y}$, with $\mathbf{Y} \in \mathbb{R}^P$. The function f can be eventually learned in a supervised fashion given a known set of $(\mathbf{X}^{(i)}, \mathbf{Y}^{(i)})$ input and output samples respectively. Here we use the notation $\mathbf{X}^{(i)}$ to refer to the i-th sample of a set of points $\mathbf{X}$. Throughout this assignment we will only cover the supervised learning case.

The f function is derived from a set of **weights values** $\mathbf{W}$ and **activation functions** g, stacked successively, forming **layers** of functions. There are many possible activation functions as seen in the lecture slides. Each layer comprises multiple unit **neurons** linked to one another via the formula,

$$a_i^l = g(\sum_{j=1}^{S^{l-1}} w_j^l . a_j^{l-1} + w_{0j}^l), \tag{1}$$

where a_i^l refers to the value of the neuron i at layer l, and S^l is the size of the l-th layer. Here we assume that the network is **fully connected** or **dense**, meaning that each **neuron** takes as input the values of all the **neurons** of the previous layer. An alternative notation using vectorization is,

$$\mathbf{A}^l = g(\mathbf{W}^l . \mathbf{A}^{l-1} + \mathbf{W}_0^l), \tag{2}$$

where $\mathbf{W}^l$ and $\mathbf{W}_0^l$ are the **weight and bias arrays** of the layer l and of dimensions $S^l \times S^{l-1}$ and $S^l \times 1$ respectively. Usually, we refer to the input layer $\mathbf{X}$ as $\mathbf{A}^0$.

For the rest of the assignment, we will use this vectorization notation represented by capital letters. This vectorization makes both the writing easier and the calculation faster as Matlab and many other programming languages heavily benefit from vectorization in terms of computation time. To simplify even further the writing we often introduce a variable $\mathbf{Z}^l$ such that $\mathbf{A}^l = g(\mathbf{Z}^l)$, where $\mathbf{Z}^l = \mathbf{W}^l . \mathbf{A}^{l-1} + \mathbf{W}_0^l$.

Computing the output of the network $f : \mathbf{X} \rightarrow \mathbf{Y}$ is refered as applying a **forward pass** or **feedforward** and corresponds to chaining the output of each layers until we reach the last one.

Activation functions

As stated previous there are plethora of activation functions g that can be used in Neural Networks. Most common ones are **sigmoid**, **hypebolic tangeant**, **Rectified Linear Unit** (ReLu) and its improved version the **Leaky ReLu**. Note that those functions are usually refered as $\mathbf{A} = g(\mathbf{Z})$, and need to be derivable w.r.t $\mathbf{Z}$. The different layers of the network can have different activation functions. However, it is recommended that the neurons of the same layer have the same activation function.

Sigmoid

The **sigmoid** is defined as,

$$\mathbf{A} = \frac{1}{1+e^{-\mathbf{Z}}} \tag{3}$$

It guarantees that the output $\mathbf{A}$ of the layer will be between 0 and 1, which is particularly useful when doing binary classification.

Hyperbolic tangent

The **hyperbolic tangent** is defined as,

$$\mathbf{A} = tanh(\mathbf{Z}) = \frac{e^{\mathbf{Z}} - e^{-\mathbf{Z}}}{e^{\mathbf{Z}} + e^{-\mathbf{Z}}} \tag{4}$$

The output of the **hyperbolic tangent** function is between -1 and 1. Mathematically, it is a shifted version of the **sigmoid** function. It turns out that **hyperbolic tangent** is better suited than the **sigmoid** function to be used in the **hidden layers** of the network.

The problem of both the **sigmoid** and the **hyperbolic tangent** is that when the absolute value of $\mathbf{Z}$ is large, the derivative is vanishing which can slow down the convergence of the network's weights. This is one of the main motivations for the introduction of the **ReLu** (Rectified Linear Unit) activation function.

ReLu

The **ReLu** is defined as,

$$\mathbf{A} = max(0, \mathbf{Z}) \tag{5}$$

ReLu is the default choice for all layers, except for the specific case of binary classifcation where the **sigmoid** function is prefered for the output layer of the network. One of the disadventages of this function is that the derivative is 0 when $\mathbf{Z} < 0$. In practice, this is not much of a problem, nevertheless this motivated the introduction of **Leaky ReLu**.

Leaky ReLu

The **Leaky ReLu** is very similar to the **ReLu**,

$$\mathbf{A} = max(k.\,\mathbf{Z}, \mathbf{Z}) \tag{6}$$

where $k = 0.01$ is usually the chosen value. k can be also see as an **hyperparameter** of the network but in practice it is common to use its default value.

Task 1: Implement the `apply_activation.m` function (4pts)

The function has the following signature:

In [ ]:
```
function [A] = forward_activation(Z, Sigma)
```

`Sigma` is a string indicating which type of activation function to use for the current layer. It can take its value from the list of previously introduced activation functions, i.e. `sigmoid`, `tanh`, `relu`, `leakyrelu`.

Batch learning

A network is usually trained over a full dataset of M samples $\{(\mathbf{X}^{(1)}, \mathbf{Y}^{(1)}), \ldots, (\mathbf{X}^{(M)}, \mathbf{Y}^{(M)})\}$. The dimensions in the calculations have to be adapted to take into account multiple samples. For example, $(\mathbf{X}, \mathbf{Y})$ will be of dimensions $N \times M$ and $P \times M$ respectively. Dimensions of $\mathbf{A}^l$ is $S^l \times M$.

Training over multiple samples is a technique referred as **batch learning**. It relies extensively on vectorized calculation and is by far more efficient in terms of computation time compared to learning sample by sample. However, when learning over thousands of samples, the matrices become simply too large for efficient calculations. Therefore, we usually perform the calculation over a subset of samples randomly selected over M.

This technique, referred as **mini-batch learning** introduces a trade-off between computation time and accuracy. Indeed, when learning over the full set of samples M, the network is guaranteed to converge to an optimal solution, if the cost function used for training is convex. This might not be the case when learning over subsets as it might converge to an optimal solution for a specific subset. Adding **mini-batch learning** method introduces another **hyperparameter** to select, the mini-batch size.

Weight initialization

At the beginning of the training, the weights $\mathbf{W}^l$ and $\mathbf{W}_0^l$ should be initialized to a some value for each layer. A naive approach could be to set them all to the same value or to zeros. However, this will result in impossible training as the network will fail to break symmetry. What this means is that each neuron will get the same value as the network is fully connected. It will be impossible, even after hours of training, to make them converge to different values. In the zero case each neuron will output the 0 value regardless of the input which is even worse.

Therefore, we usually set them to random values, such that each neuron performs a different computation. The range of values is arbitrary but a common approach is to initialize them with random numbers drawn from $\mathcal{N}(0, 1)$.

There are other possible initialization such as `xavier` or `he` initialization techniques. See http://deepdish.io/2015/02/24/network-initialization/ for more information on those techniques.

Task 2: Implement `initialize_weights.m` function (2pts)

This function has the following signature:

In [ ]:
```
function [W, W0] = initialize_weights(LayerSizes, type)
```

The `LayerSizes` is a cell array providing a description of the network by giving the number of neurons per layer. Note that it also provides the size of the input layer $\mathbf{X} = \mathbf{A}_0$, therefore, the size of `LayerSizes` is $L+1$, with L the total number of layers of the network. Thus, you need to take that into acount when filling the weights matrices.

This function should output the layer weights `W` and bias `W0` as cell arrays where each cell contains matrices of weights and biases for the l-th layer. `type` corresponds to the desired type of initialization. It can take the values `random` or `zeros`

Implementation hint: Useful functions are `randn` and `zeros` for initialization and `cell` and `{}` for cell arrays

Feedforward

The learning in Neural Networks uses gradient descent optimization algorithm and can be summarized as two main functions: **feedforward** and **backpropagation**. The feedforward method goal is to apply a **forward pass** on the network, i.e. estimating the output value $\mathbf{Y}$ from a given input $\mathbf{X}$. We recall that the values $\mathbf{A}^l$ and $\mathbf{Z}^l$, for each layers l, are computed as,

$$\mathbf{Z}^l = \mathbf{W}^l . \mathbf{A}^{l-1} + \mathbf{W}_0^l \tag{7}$$
$$\mathbf{A}^l = g(\mathbf{Z})^l \tag{8}$$

Task 3: Implement the `forward_pass.m` function (4pts)

This function has the following signature:

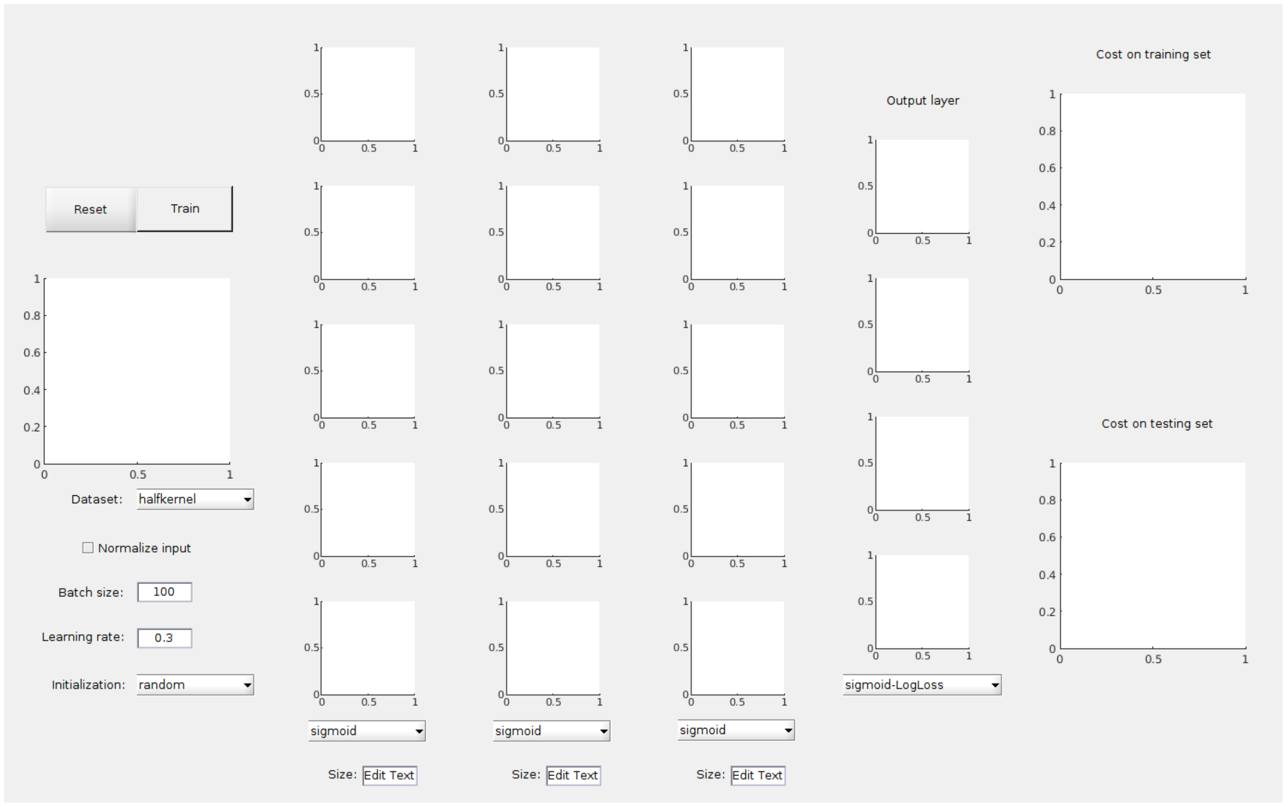
In [ ]:
```
function [YHat, A, Z] = forward_pass(X, W, W0, Sigmas)
```

It takes as input the input vector `X` and the parameters of the network `W` and `W0` weights and bias array respectively. Both are cell arrays of dimension L, number of layers of the network. Each of their cell contains a matrix whose dimensions depends on the number of neurons per layer. `Sigmas` is also a cell array of dimension L containing the strings that represent the type of activation functions for each layer.

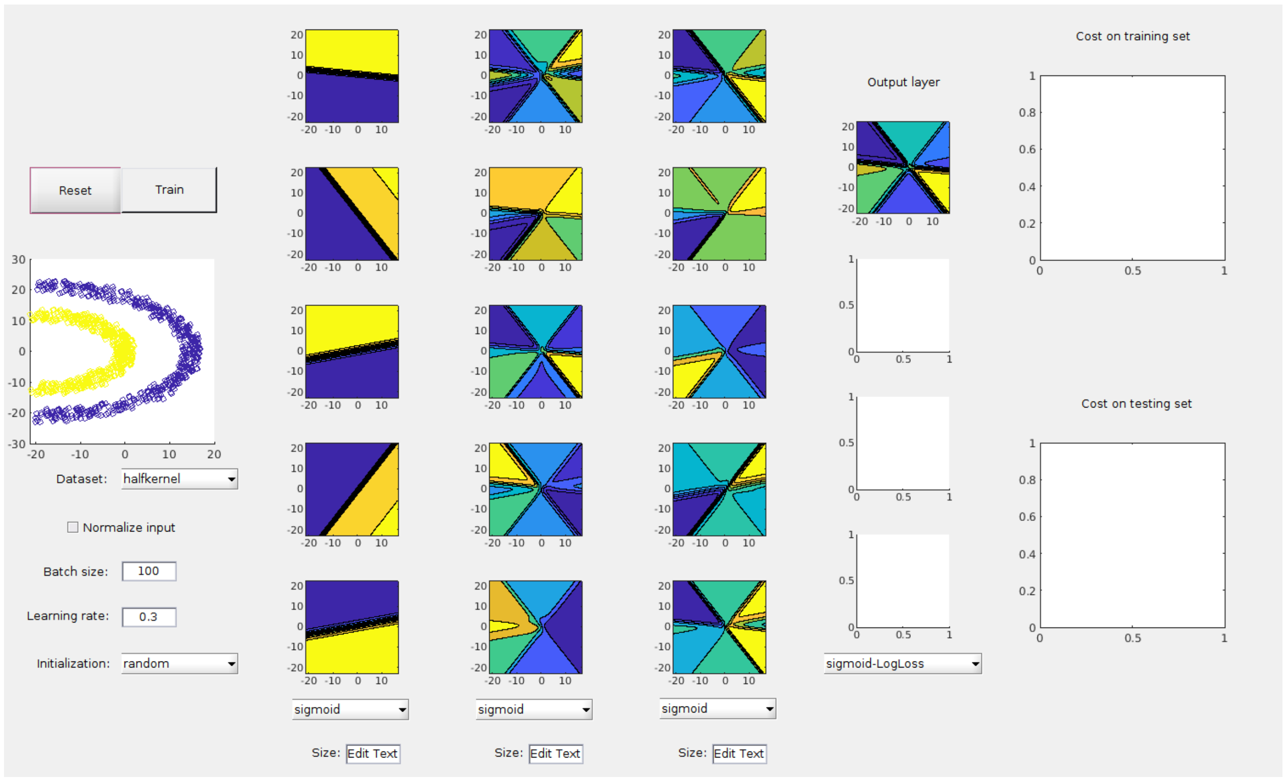
Implementation hint: Useful function `forward_activation` .

Visualizing the initialization

A graphical interface has been implemented in Matlab to help you visualizing the effect of the learning process. Once all previous functions have been implemented, you can start the GUI by running the file `gui.m` . This will open the following window:



Selecting a dataset from the dataset list will initialize the weights of the network using the initialization function selected:



Dataset: halfkernel

☐ Normalize input

Batch size: 100

Learning rate: 0.3

Initialization: random

Each plot represents the output of the activation function of each neuron.

The next step is to train the network to converge to the desired output.

Backpropagation

Backpropagation applies a **backward pass**, i.e. given the expected output value $\mathbf{Y}^d$ of an input $\mathbf{X}$, it calculates the error $E(\mathbf{Y}^d, \mathbf{Y})$ and propagates it back to each layers of the network using the derivative chain rule. Here $\mathbf{Y}^d$ refers to the ground truth value of the training set and $\mathbf{Y}$ to the output of the network. The derivatives of the weights at each layer are given by,

$$\frac{\delta E}{\delta \mathbf{Z}^l} = (\mathbf{W}^{l+1})^T \cdot \frac{\delta E}{\delta \mathbf{Z}^{l+1}} \odot \frac{\delta \mathbf{A}^l}{\delta \mathbf{Z}^l} \tag{9}$$

$$\frac{\delta E}{\delta \mathbf{W}_0^l} = \frac{1}{M} \sum_{i=1}^{M} \frac{\delta E}{\delta \mathbf{Z}_i^l} \tag{10}$$

$$\frac{\delta E}{\delta \mathbf{W}^l} = \frac{1}{M} \frac{\delta E}{\delta \mathbf{Z}^l} \cdot (\mathbf{A}^{l-1})^T \tag{11}$$

where $\odot$ corresponds to the **element-wise** product, and $\frac{\delta \mathbf{A}^l}{\delta \mathbf{Z}^l}$ are the derivatives of the activation functions at layer l. For the last layer, output of the network, we have,

$$\frac{\delta E}{\delta \mathbf{Z}^L} = \frac{\delta E}{\delta \mathbf{A}^L} \cdot \frac{\delta \mathbf{A}^L}{\delta \mathbf{Z}^L} \tag{12}$$

As seen, to compute those derivatives, one needs to recursively calculate them starting from the ouptut layer of the network. This is really the meaning of **propagating back** the error from the last layer.

Task 4: Implement `backward_activation.m` function (4pts)

The function has the following signature:

In [ ]:
```
function [dZ] = backward_activation(Z, Sigma)
```

It is similar to the `forward_activation.m` function but computes the derivative of the activation functions evaluated in `Z`.

Cost functions

The **backpropagation** propagates the error to each layer of the network. Therefore, choosing the correct **error function** is a crucial component. One thing to remember is that the **error function** should be differentiable and, preferably, convex to ensure reaching a global minimum.

For the case of binary classification where the output of the network can take either the value 0 or 1, the cost function usually considered is the **logarithmic loss**, coupled to a **sigmoid** activation function for the last layer,

$$E = -\frac{1}{M}\sum_{i=1}^{M}(Y^{d(i)}.log(Y^{(i)}) + (1 - Y^{d(i)}).log(1 - Y^{(i)})) \tag{13}$$

The intuition behind this cost function is if $Y^{d(i)} = 1$, then $E = -log(Y^{(i)})$. To minimize E, $Y^{(i)}$ will then be selected as very large, i.e. $Y^{(i)} = 1$. Inversely, if $Y^{d(i)} = 0$, then $E = -log(1 - Y^{(i)})$ and $Y^{(i)}$ will converge towards 0.

Task 5: Implement the `cost_function.m` function (2pts)

This function has the following signature:

In [ ]:
```
function [C] = cost_function(Y, Yd, type)
```

with `type` a string representing the cost function to calculate. For the **logarithmic loss** it takes the value `LogLoss`. Later in the assignment, we will modify this function to add other cost functions.

Task 6: Implement the cost_derivative.m function (2pts)

This function has the following signature:

In [ ]:
```
function [dZ] = cost_derivative(Y, Yd, typeCost, typeLayer)
```

The two inputs `typeCost` and `typeLayer` correspond to the type of cost function and last activation layer selected respectively. For the binary case it will be `LogLoss` and `Sigmoid`. This function will be modified later to include other types of network configurations.

The output value `dZ` corresponds to the derivative of the cost function $\frac{\delta E}{\delta \mathbf{Z}^L}$.

Task 7: Implement the `backward_pass.m` function (4pts)

This function has the following signature:

In [ ]:
```
function [dZ, dW, dW0] = backward_pass(dE, W, A, Z, Sigmas)
```

The input `dE` corresponds to the derivative of the cost function $\frac{\delta E}{\delta \mathbf{Z}^L}$, output of the `cost_derivative` function. `A` and `Z` are cell arrays of size $L+1$ and L respectively and are the results of the forward pass. `Sigmas` is again a cell array of size L containing the type of activation functions at each layer.

Implementation hint: Useful function `backward_activation`.

Update rule

The values of the **weights** are updated according to the **gradient descent** rule,

$$\begin{aligned}\mathbf{W}^l &= \mathbf{W}^l - \eta \frac{\delta E}{\delta \mathbf{W}^l} \\ \mathbf{W}_0^l &= \mathbf{W}_0^l - \eta \frac{\delta E}{\delta \mathbf{W}_0^l},\end{aligned} \tag{14}$$

where η is an hyperparameter known as the **learning rate**. Note that different solution can be used to update the weights by using for example a decay of the learning rate but this goes beyond the scope of this assignment. If you are interested in learning more about it you can have a look at this article.

Task 8: Implement the `update_weights.m` function (2pts)

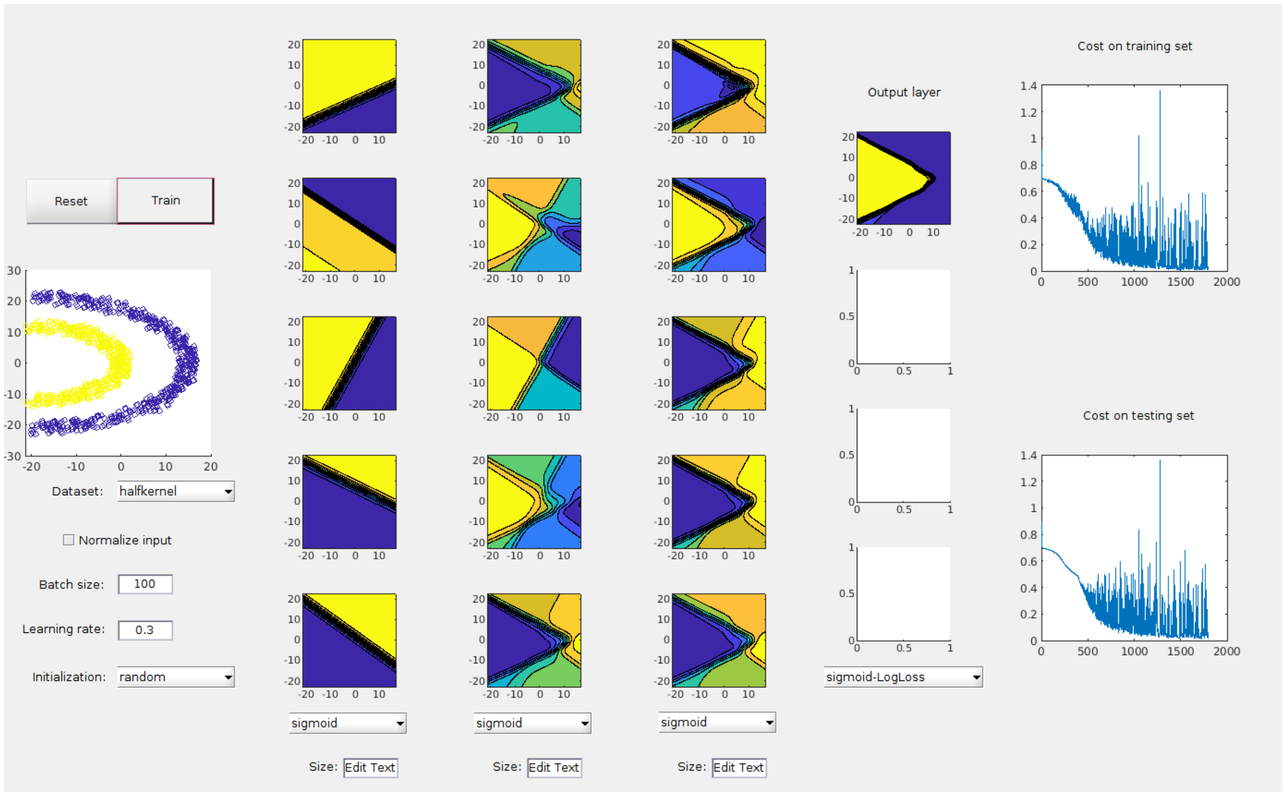
This function has the following signature:

```
In [ ]: function [newW, newW0] = update_weights(W, W0, dW, dW0, eta)
```

`eta` is the value of the learning rate η. This function outputs the new weights after backprogation.

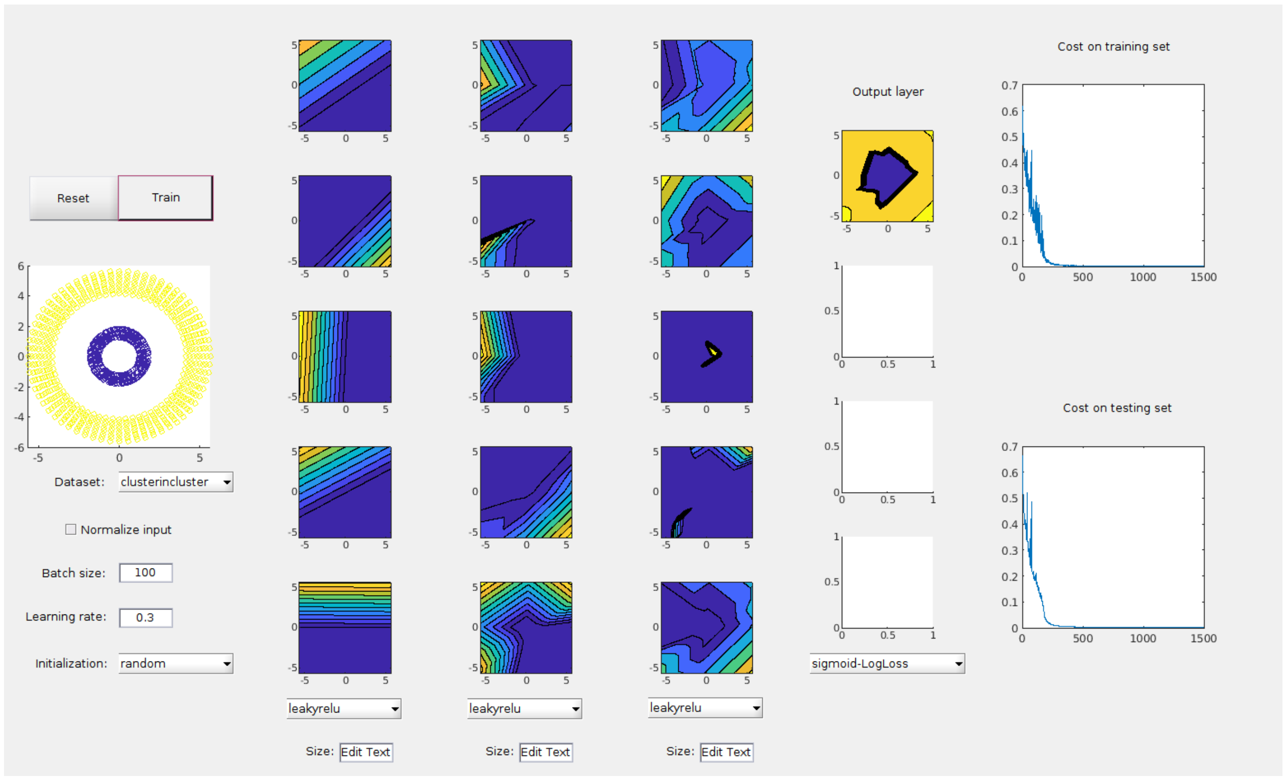
Visualizing the learning process

Clicking the `Train` button will start the learning process that should slowly converge to the desired results:



Dataset: halfkernel
Normalize input
Batch size: 100
Learning rate: 0.3
Initialization: random

You can select different activation functions for each layer and observe the results:



Dataset: clusterincluster

☐ Normalize input

Batch size: 100

Learning rate: 0.3

Initialization: random

If you try to change the output and cost function of the output layer, you will obtain an error as some more implementation is necessary to handle multi-class classification.

Part2: Multi-class classification

Multi-class classification is very similar to binary classification but requires changing the activation function of the output layer and its associated cost function. First we need to also change the representation of the classes.

One could see the training of a network for multiple classification as a regression problem where the output of the network is trained to fit the labels from the training set, i.e. it should be an integer number corresponding to the class associated to the label. In practice this is not desired as activation functions of the Neural Networks are usually continuous to facilitate the convergence, meaning that forcing them to output a proper integer is not realistic.

Instead, it is common practice to apply a transformation to the labels in the set.

One-hot encoding

One-hot encoding is a common transformation to consider and consists of representing each label by a vector of dimension C where C is the total number of classes in the label set. The index of the class corresponding to the label will then be assigned the value 1 one the other will have a 0 value. For example, a sample belonging to class 3 in a dataset with 4 classes is represented as $[0, 0, 1, 0]$.

Functions for one hot encoding and decoding are provided in the `utils` folder.

Softmax activation function

The activation function of the last layer has to be modified to take into account the one-hot encoding described previously. But instead of simply providing a vector of 0s and 1s, it can be better to consider the last layer as a probability function, i.e. the probability distribution of a sample to belong to a specific class. Considering our 4-class dataset example, if the output corresponding to one sample is $[0.1, 0.1, 0.8, 0]$, this can be interpreted as the sample has 10% probability to be of class 1 and 2 and 80% probability of being of class 3.

As a probability function is continuous this representation facilitates the convergence of the network and also provides an insight on the certainty of the prediction. If the output is $[0.25, 0.25, 0.25, 0.25]$ the prediction is completly uncertain.

To perform that operation, one need to change the activation function of the last layer as if we keep sigmoid activation functions, there is no guarantee that the output vector will be normalized. For that we introduce the **softmax** activation function as,

$$softmax(\mathbf{Z}) = \frac{e^{\mathbf{Z}}}{\sum e^{\mathbf{Z}}}. \tag{15}$$

There is a possibility for the **softmax** function to be numerically unstable when reaching infinity. To prevent that, one can add a constant δ in the calculation such that,

$$softmax(\mathbf{Z}) = \frac{e^{\mathbf{Z}-\delta}}{\sum e^{\mathbf{Z}-\delta}}. \tag{16}$$

where δ is usually selected as $\delta = max(\mathbf{Z})$ of all the neurons in the last layer. Be careful with the dimensions as you want a different δ value for each of the M samples.

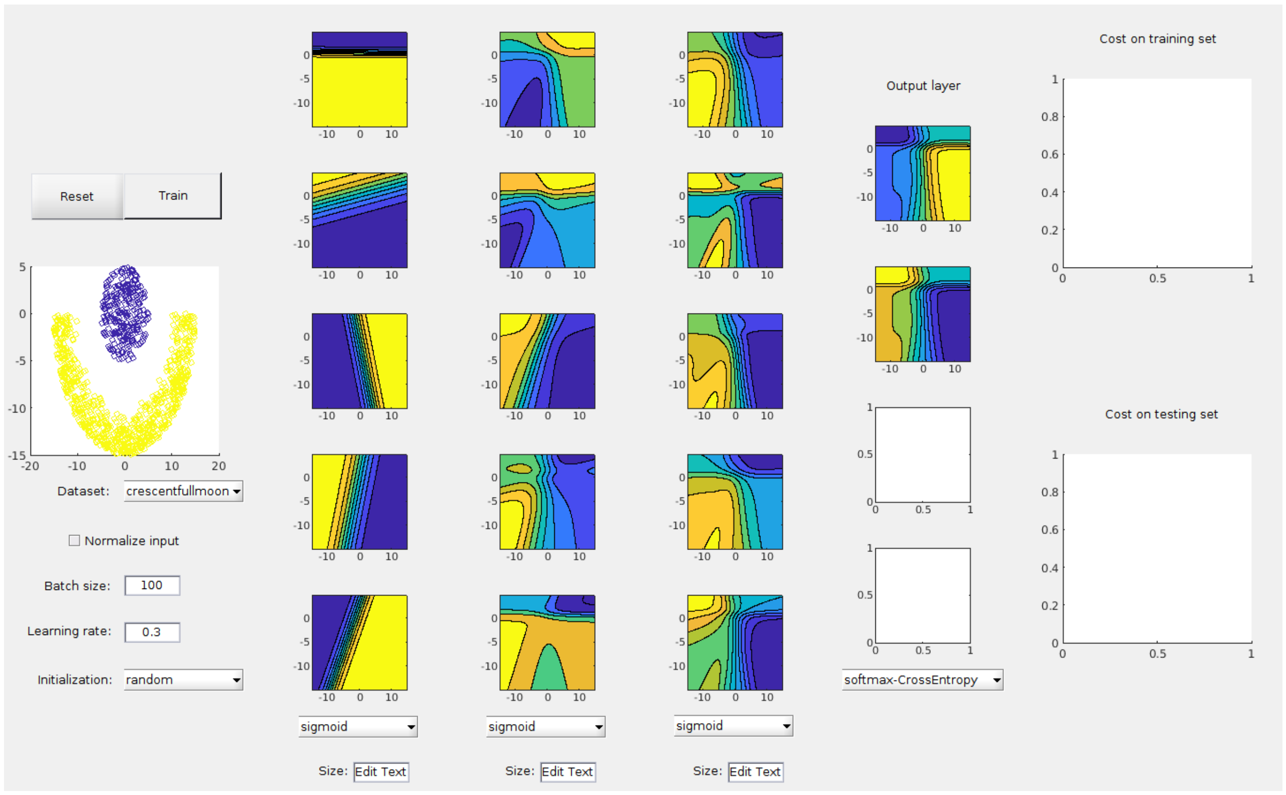
With this activation function, the ouptut vector is guaranted to be normalized and can, therefore, be seen as a probability distribution.

Task 9: Add the `softmax` calculation in `forward_activation.m` (2pts)

You have to modify the function `forward_activation.m` to add the `softmax` activation function calculation described above.

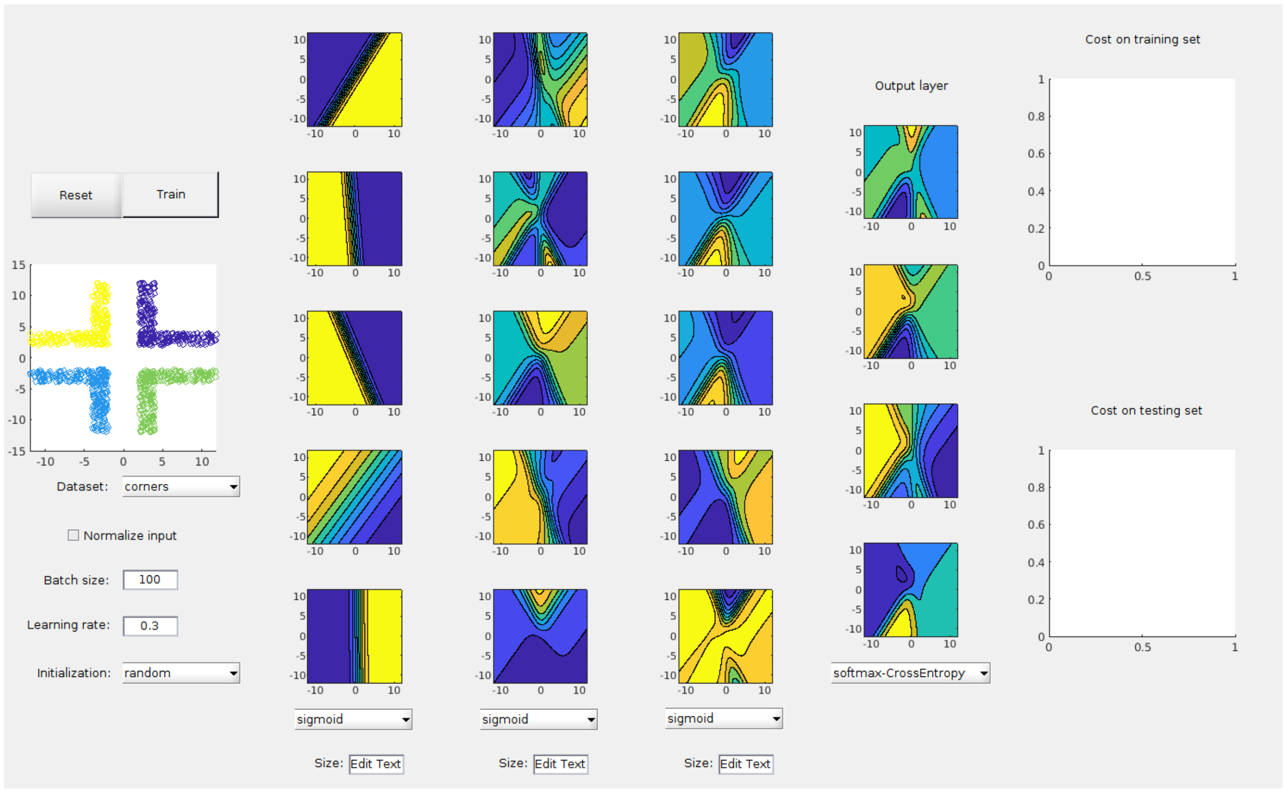
Visualizing multi-class dataset

You can now select the `softmax-CrossEntropy` in the list below the output layer. This will initialize randomly the network to handle multi-class output. For a 2 classes dataset this will result in the output layer being of dimension $2 \times M$:



Dataset: crescentfullmoon
Normalize input
Batch size: 100
Learning rate: 0.3
Initialization: random
sigmoid
sigmoid
sigmoid
softmax-CrossEntropy

For a 4 classes dataset such as the `corners` this gives:



Dataset: corners

Batch size: 100

Learning rate: 0.3

Initialization: random

sigmoid sigmoid sigmoid

softmax-CrossEntropy

Be careful that trying to train the network will give an error unless you change the cost function associated to the last layer.

Cross entropy cost function

Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events. It is composed of two main components, the **Kullback-Leibler divergence** and the **Entropy** measure.

Kullback-Leibler divergence

Given two probability distribution P and Q the **Kullback-Leibler divergence** defines the distance between the two distributions as:

$$D_{KL}(P||Q) = -\sum_{i=1}^{N} P(x^i)log(\frac{Q(x^i)}{P(x^i)}) \tag{17}$$

This measure will ensure that the probability distribution Q output of the network will converges toward the desired probability P.

Entropy

Entropy is the number of bits required to transmit a randomly selected event from a probability distribution. A skewed distribution has a low entropy, whereas a distribution where events have equal probability has a larger entropy.

Therefore, one can see the entropy as a measure of uncertainty defined as,

$$H = -\sum_{i=1}^{N} P(x^i) log(P(x^i)) \tag{18}$$

if $P(x_i)$ is very uncertain, i.e. the probabilities are balanced the entropy is high. Inversely, if the probabilities are skewed towards a single class then the entropy is low. Minimizing the entropy will, therefore, minimize the uncertainty for each samples.

Cross entropy

The cross entropy is finally the sum of both the **Kullback-Leibler divergence** and the **Entropy** measures, defined as.

$$H(P, Q) = -\sum_{i=1}^{N} P(x^i) log(Q(x^i)) \tag{19}$$

This metric ensures that the approximation Q of a distribution P will be close while minimizing its uncertainty. Applied to the output of the network, the **cross entropy loss** function is defined as,

$$E = -\frac{1}{M} \sum_{i=1}^{M} \sum_{j=1}^{N} Y_j^{d(i)} . log(Y_j^{(i)}) \tag{20}$$

Task 10: Add the `CrossEntropy` calculation in `cost_function.m` (2pts)

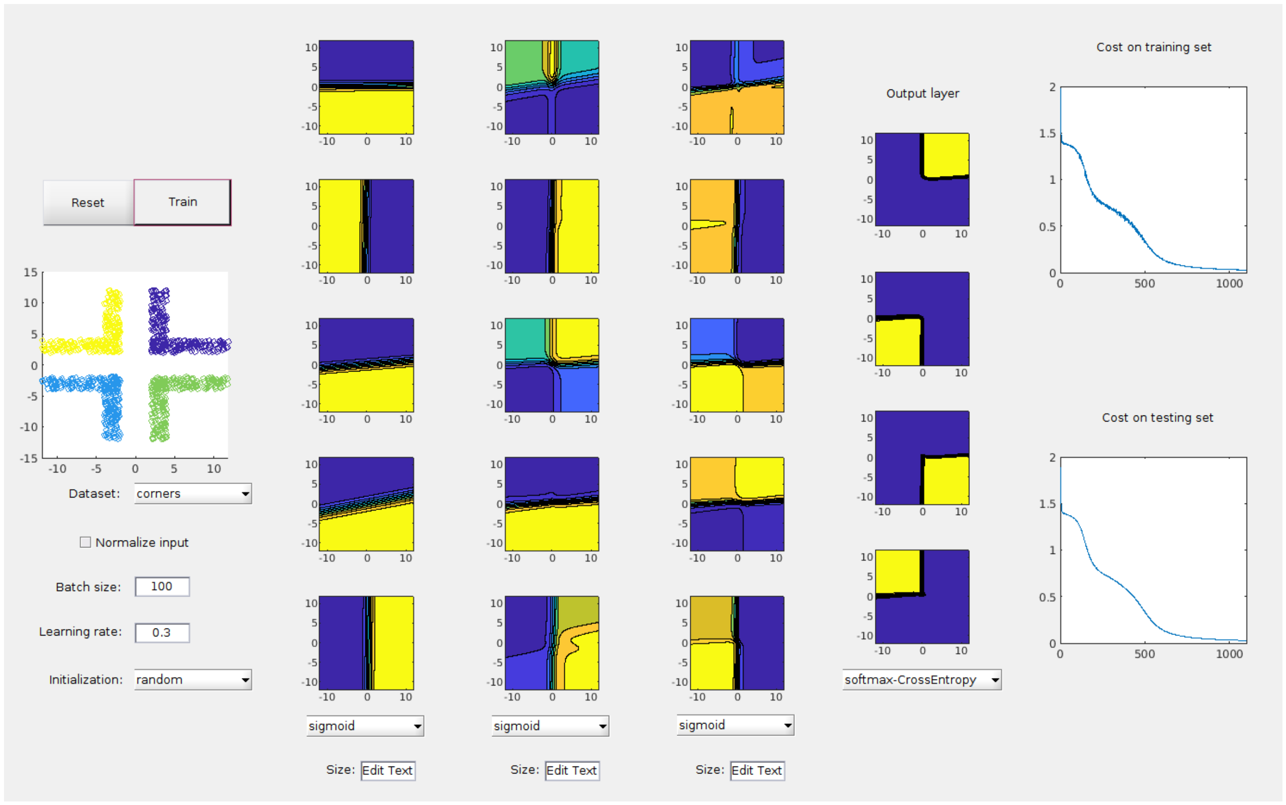
You have to modify the function `cost_function.m` to add the **cross entropy loss** function calculation. Note that type will take the value `CrossEntropy` .

Task 11: Add the `CrossEntropy` derivative in `cost_derivative.m` (2pts)

You have to modify the function `cost_derivative.m` to add the cross entropy loss derivative calculation.

Visualizing the training

Once all the functions have been implemented you can train the network with a 4 class dataset such as `corners` which should converge to the desired output.



Dataset: corners

Normalize input

Batch size: 100

Learning rate: 0.3

Initialization: random

sigmoid sigmoid sigmoid

softmax-CrossEntropy

Size: Edit Text Size: Edit Text Size: Edit Text

Conclusion

Congratulation for making it to the last assignment of the Machine Learning Proframming course. We hope you have found that course useful, learned interesting new skills and we wish you wonderful holidays and a great year 2022!