Machine Learning Programming
Assignment 6: Neural Networks

Professors: Baptiste Busch, Aude Billard
Assistants: Laila El Hamamsy, Matthieu Dujany
and Victor Faraut
Contacts:
baptiste.busch@epfl.ch, aude.billard@epfl.ch
laila.elhamamsy@epfl.ch, matthieu.dujany@epfl.ch, victor.faraut@epfl.ch

Winter Semester 2018

Introduction

In this optional practical, you will implement implement the basics of neural networks, including activation functions and cost functions.

Submission Instructions

Deadline: January 8, 2019 @ 6pm. This is a bonus assignment. **Procedure:** From the course Moodle webpage, the student should download and extract the .zip file named `TP6-NN-Assignment.zip` which contains the following files:

Code
`cost_function.m`
`cost_derivative.m`
`apply_activation.m`
`initialize_weights.m`
`gui.m`
`gui.fig`
`MyNeuralNetwork.m`

Assignment Instructions

The assignment consists on implementing the blue colored MATLAB functions from scratch. For the Part 2 of the assignment you can modify any of the black function at your convenience so be sure to include them as well in your submission.

Once you have tested your functions, you can submit them as a .zip file with the name: `TP6-NN-SCIPER.zip` on the submission link in the Moodle webpage. Your submission archive should contain ONLY the following:

🟥 `TP5-GMMApps-Assignment-SCIPER`
└── 🟦 `Code`

DO NOT upload `utils` directory.

NOTE: This assignment is optional. It will be rewarded with some bonus points.

1 Neural Network Theory

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

1.1 Feedforward and backpropagation

As a reminder, the core principle of Neural Networks is, given an input $X \in \mathbb{R}^N$ to define a mapping $f : X \to Y$, with $Y \in \mathbb{R}^P$. The function f can be eventually learned in a supervised fashion given a known set of $(X^{(i)}, Y^{(i)})$ input and output samples respectively[1]. Throughout this assignment we will only cover the supervised learning case.

The f function is derived from a set of `weights values` 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,

$$a_i^l = g(\sum_{j=1}^{S^{l-1}} w_j^l . a_j^{l-1} + w_0^l), \tag{1}$$

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 `neurons` takes as input the values of all the `neurons` of the previous layer. An alternative notation using vectorization is,

$$A^l = g(W^l . A^{l-1} + W_0^l), \tag{2}$$

where W^l and W_0^{l}[2] are of dimensions $S^l \times S^{l-1}$ and $S^l \times 1$ respectively. Usually, we refer to the input layer X as A^0. This notation makes both the writing easier and the calculation faster[3]. For the rest of the assignment we will mainly use the vectorized notation. To simplify even further the writing we often use the notation $A^l = g(Z^l)$, where $Z^l = W^l . A^{l-1} + W_0^l$.

The learning in Neural Networks 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 Y from a given input X. The `backpropagation` applies a backward pass, i.e. given the expected output value Y^d of an input X, it calculates the error $E(Y^d, Y)$ and propagates it back to each layers of the network. The values of the `weights` are updated according to a specific rule. For example, using the `gradient descent` update rule, `weigths` are updated as,

[1]Here we use the notation $X^{(i)}$ to refer to the i-th sample of a set of points X

[2]Sometimes, the `bias` value W_0 is note b. In the code we use the b notation.

[3]Matlab heavily benefits from vectorization in terms of computation time.

$$
\begin{aligned}
W^l &= W^l - \eta \frac{\delta E}{\delta W^l} \\
W_0^l &= W_0^l - \eta \frac{\delta E}{\delta W_0^l},
\end{aligned}
\tag{3}
$$

where $\frac{\delta E}{\delta W^l}$ is calculated by using the derivative chain rule.

1.2 Cost functions

The `backpropagation` propagates the error to each layer of the network. Therefore, choosing the correct `error function` is a crucial component. During the lecture you have seen three different ones but there exists many others. One thing to remember is that the `error function` should be derivable and, preferably, convex.

1.3 Batch learning

The equations (2-3) are defined for a single sample $(X^{(i)}, Y^{(i)})$. However, a network is usually trained over a full dataset of M samples $\{(X^{(1)}, Y^{(1)}), \ldots, (X^{(M)}, Y^{(M)})\}$. In this case, the dimensions in the calculations have to update to take into account multiple samples. For example, (X, Y) will be of dimensions $N \times M$ and $P \times M$ respectively. Dimensions of A^l will also change to $S^l \times M$. The definition of equation 2 is not impacted by this change and W^l will keeps its dimensions, i.e. $S^l \times S^{l-1}$. The case of W_0^l is a bit more specific as it should be of size $S^l \times M$ for the calculation to be correct. However, many moderns mathematical library are perfectly fine with this notation and simply repeat the values of the first column M times prior to perform the calculation. Therefore, W_0^l also keeps its dimension of $S^l \times 1$. However, the update rule of equation 3 needs to be updated to,

$$
\begin{aligned}
W^l &= W^l - \frac{\eta}{M} \frac{\delta E}{\delta W^l} \\
W_0^l &= W_0^l - \frac{\eta}{M} \frac{\delta E}{\delta W_0^l},
\end{aligned}
\tag{4}
$$

Training over multiple samples is a technique referred as `batch learning`. As it relies extensively on vectorized calculation it 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`, the mini-batch size.

2 Part 1: Implementation

The theory detailed in section 1 has been implemented in the a Matlab class `MyNeuralNetwork.m`. You should carefully read it to check if you correctly understand how it works and how each function is implemented. A graphical interface has also been implemented around the class to display the learning process. The interface comprises two files `gui.m` and `gui.fig`. You should also open and check them. This assignment is not here to teach you how to create a graphical interface in Matlab, therefore, only the basic concepts will be explained if required. You can check the official documentation at `https://ch.mathworks.com/discovery/matlab-gui.html`.

In this assignment, we will not ask you to implement the `feedforward` and `backpropagation` functions, that are already provided in the `MyNeuralNetwork.m`. However, you will have to implement the `activation functions`, `weight initialization`, and `cost functions`. In the next sections we will detail each of them.

2.1 Activation functions

The first elements to implement are the activation functions. Each activation function is used in both the forward and the backward pass. During the backward pass, the gradient of the activation function $\frac{\delta A^l}{\delta Z^l}$ is required. There are three main activation functions that you will have to implement: Sigmoid, tanh, and ReLU. Remember to implement both forms, the normal form and its derivative. For the sigmoid, for simplicity you will consider the slope as $D = 1$. There are many other activation functions that you can also try such as RBF or Leaky ReLU.

Test implementation

Implement the activation functions in `apply_functions.m`. The `Sigma` gives you the type of activation function, while `isForward` tells you if your are calling the function for the forward or the backward pass. At this stage, you cannot test your implementation visually using the graphical interface. However, here are a few values all tested with a seed of 42

```
>>rng(42)
>>apply_activation(randn(3,4), 'sigmoid', true)

 ans =

     0.3686    0.5836    0.2146    0.6803
     0.7042    0.2697    0.7145    0.3563
     0.7263    0.3721    0.5440    0.8635

>>rng(42)
>>apply_activation(randn(3,4), 'sigmoid', false)

 ans =

     0.2327    0.2430    0.1685    0.2175
     0.2083    0.1970    0.2040    0.2293
     0.1988    0.2336    0.2481    0.1179

>> rng(42)
>> apply_activation(randn(3,4), 'tanh', true)

 ans =

    -0.4917    0.3251   -0.8611    0.6382
     0.7000   -0.7599    0.7247   -0.5310
     0.7513   -0.4802    0.1748    0.9512

>> rng(42)
>> apply_activation(randn(3,4), 'tanh', false)
```

```

ans =

    0.7583    0.8943    0.2586    0.5927
    0.5100    0.4225    0.4749    0.7181
    0.4355    0.7694    0.9694    0.0952

>> rng(42)
>> apply_activation(randn(3,4), 'relu', true)

ans =

         0    0.3374         0    0.7552
    0.8672         0    0.9174         0
    0.9760         0    0.1766    1.8444

>> rng(42)
>> apply_activation(randn(3,4), 'relu', false)

ans =

  3  4  logical array

   0   1   0   1
   1   0   1   0
   1   0   1   1
```

If you want to add more activation functions, simply add a new type. You can then modify the gui by entering `guide` in Matlab console. Select `Open existing GUI` and select the file `gui.fig`. This will open a template of the graphical interface, as shown in figure 1 where you can add items and change existing components.

By double clicking on each of the drop lists in the highlighted area you can change the properties to add more elements. This will open another menu on which you should click as shown on the properties shown in figure 2.

2.2 Weight initialization

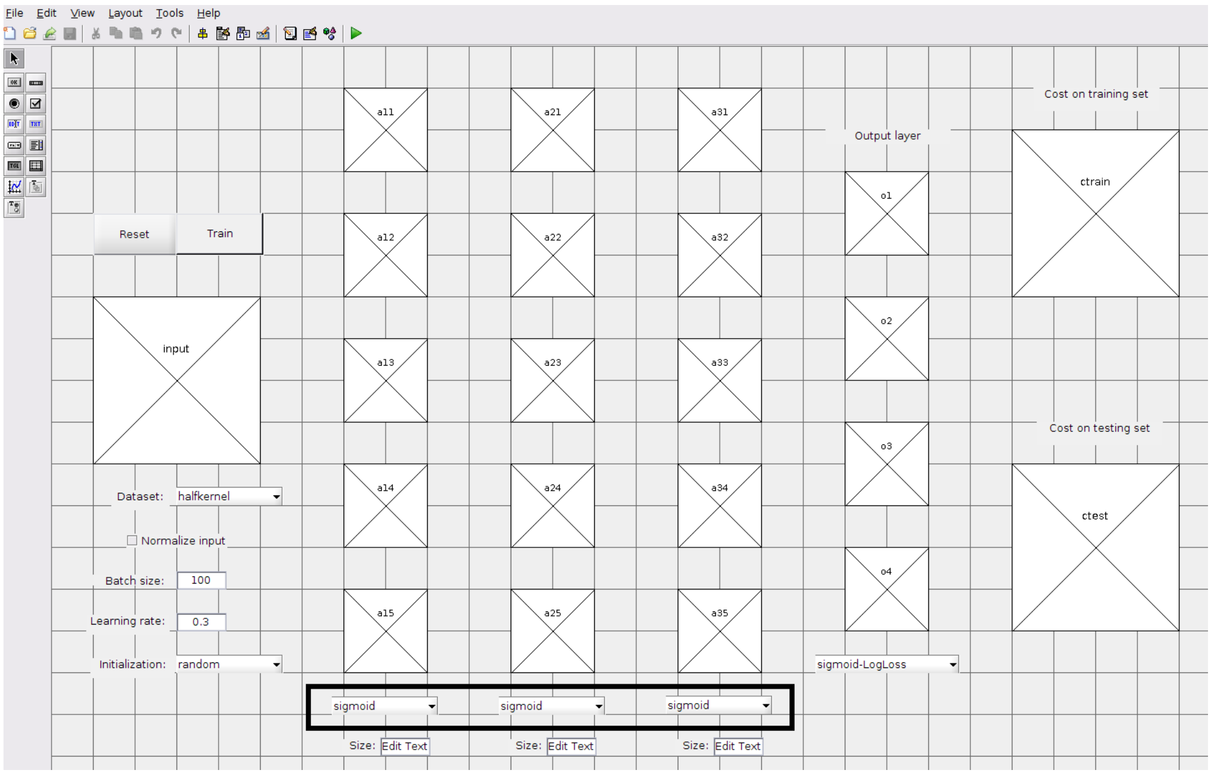
At the beginning of the training, the weights W^l and W_0^l should be initialized to a given value for each layers. A naive approach could be to set them all to the same value or to zeros. However, this will result in an impossible training as the network will fail to break symmetry. What this means is that each neurons 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[4].

Test implementation

You will have to implement the initialization of the weights in the `initialize_weights.m` function. This function takes the network as input and a type of initialization. You will have to, at least, implement the `zeros` and the `random` cases. Dimensions of each layers are contained

[4]See `http://deepdish.io/2015/02/24/network-initialization/` for more information on those techniques



Dataset: halfkernel

Batch size: 100

Learning rate: 0.3

Initialization: random

sigmoid sigmoid sigmoid

sigmoid-LogLoss

Figure 1: The graphical interface creation tool. The black rectangle shows the list to add more activation functions

in the `Layers` attribute of `MyNeuralNetwork` class, which is a cell array. Remember that the first element of `Layers` array is the input dimension while the last one is the output dimension.

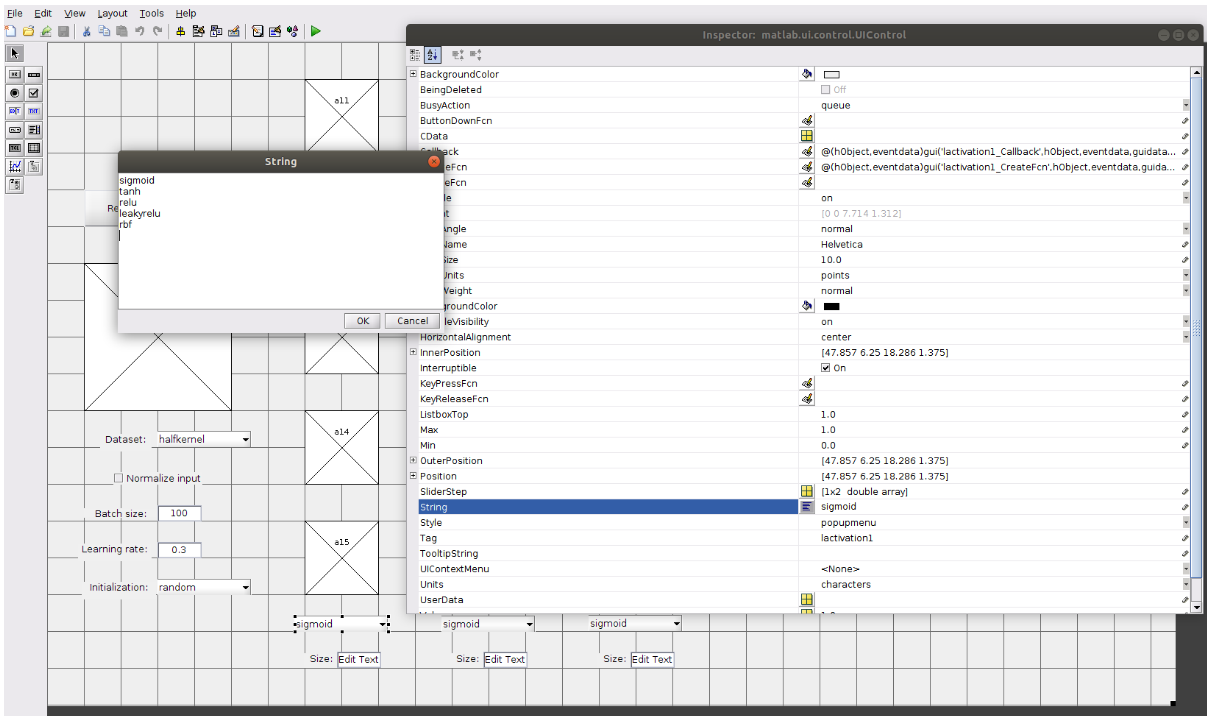
Once you have implemented both the activation functions and the weight initialization you can start the graphical interface by opening the file `gui.m` and clinking on `Run`. A window should open with blank graphs. Select a dataset on the `Dataset` list and you should obtain a figure similar to 3

2.3 Cost functions

Cost functions are needed for training. The provides the error necessary for backpropagation. Similarly to the activation functions, you will need to implement both the normal function and its derivative. However, concerning the derivative you will already apply one level of chain rule to already calculate $\frac{\delta E}{\delta Z^L}$ where L corresponds to the index of the last layer. Remember that,

$$\frac{\delta E}{\delta Z^L} = \frac{\delta E}{\delta A^L} \cdot \frac{\delta A^L}{\delta Z^L}. \tag{5}$$

You will have to implement both the Logarithmic Loss `LogLoss` for binary classification and the Cross Entropy `CrossEntropy`. Be careful that both errors are calculated over multiple samples and not single ones. Therefore, we refer to them as cost function instead of error functions.


File Edit View Layout Tools Help
Inspector: matlab.ui.control.UIControl
String
OK
Cancel
a11
a14
a15
Re

```
sigmoid
tanh
relu
leakyrelu
rbf
```

Dataset: halfkernel

☐ Normalize input

Batch size: 100

Learning rate: 0.3

Initialization: random

sigmoid — Size: Edit Text | sigmoid — Size: Edit Text | sigmoid — Size: Edit Text

BackgroundColor	▭
BeingDeleted	☐ Off
BusyAction	queue
ButtonDownFcn	
CData	
Callback	@(hObject,eventdata)gui('lactivation1_Callback',hObject,eventdata,guidata...
eFcn	@(hObject,eventdata)gui('lactivation1_CreateFcn',hObject,eventdata,guida...
eFcn	
le	on
t	[0 0 7.714 1.312]
Angle	normal
Name	Helvetica
Size	10.0
Units	points
Weight	normal
groundColor	■
leVisibility	on
HorizontalAlignment	center
InnerPosition	[47.857 6.25 18.286 1.375]
Interruptible	✓ On
KeyPressFcn	
KeyReleaseFcn	
ListboxTop	1.0
Max	1.0
Min	0.0
OuterPosition	[47.857 6.25 18.286 1.375]
Position	[47.857 6.25 18.286 1.375]
SliderStep	[1x2 double array]
String	sigmoid
Style	popupmenu
Tag	lactivation1
TooltipString	
UIContextMenu	<None>
Units	characters
UserData	

Figure 2: The property editor for the drop lists

To use the Cross Entropy you will also need to implement the softmax layer in `apply_activation.m` function. Recall that,

$$softmax(Z^L) = \frac{e^{Z^L}}{\sum e^{Z^L}}. \tag{6}$$

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

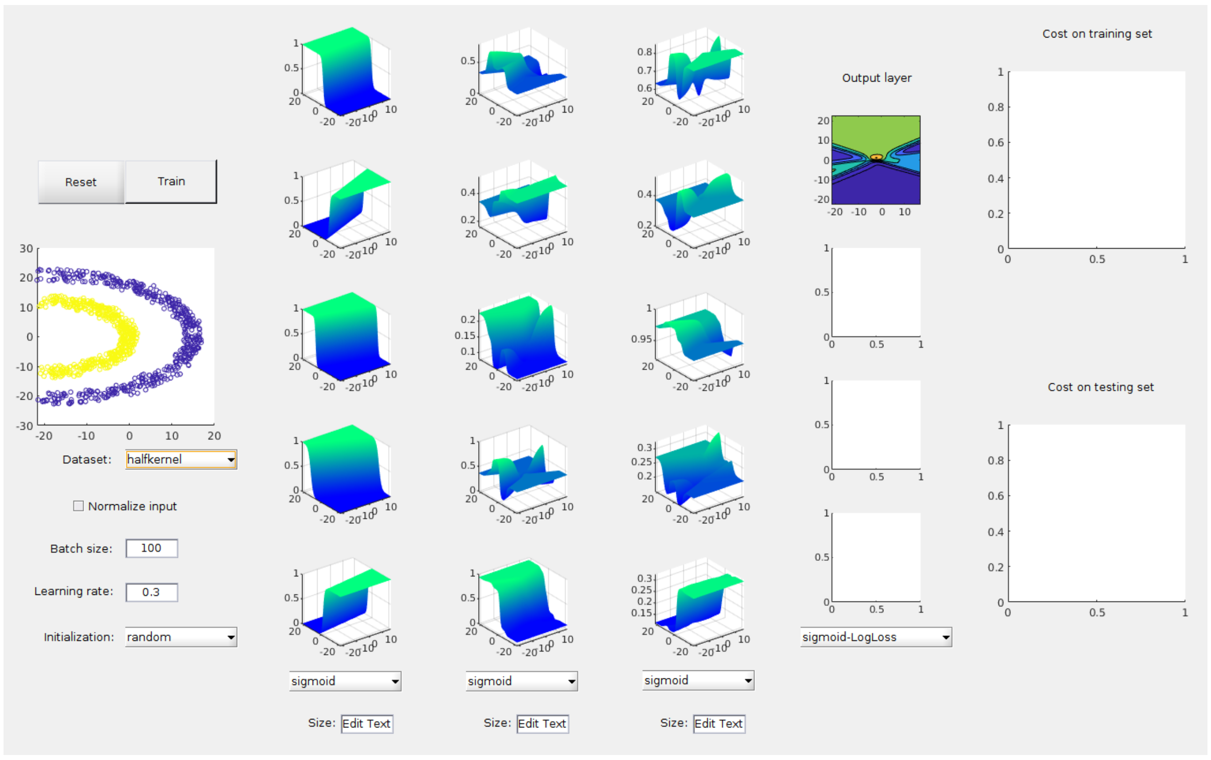
$$softmax(Z^L) = \frac{e^{Z^L - C}}{\sum e^{Z^L - C}}. \tag{7}$$

Usually this constant is selected as $C = max(Z)$ of all the neurons in the last layer. Be careful with the dimensions as you want a different C value for each sample. In the case of the softmax you do not need to implement its derivative as it will only be used as the last layer and derivative of the last layer is calculated in the `cost_derivative.m` file.

Test implementation

You have to implement the cost functions in two separate files `cost_function.m` and `cost_derivative.m`. `cost_function.m` contains the forward implementation of the cost function with the variable `type` indicating which one to apply. `cost_derivative.m` contains the derivative of the cost function w.r.t the last layer $\frac{\delta E}{\delta Z^L}$.

Once both functions are implemented you can start training your network and check that it correctly converges to the desire boundaries for classification. To help you make a careful check here are some values of the cost function with seed 42.


Dataset: halfkernel

Normalize input

Batch size: 100

Learning rate: 0.3

Initialization: random

sigmoid

sigmoid

sigmoid

sigmoid-LogLoss

Size: Edit Text

Size: Edit Text

Size: Edit Text

Figure 3: The graphical interface with `random` weight initialization

```
>> rng(42)
>> cost_function(zeros(1,4), rand(1,4), 'LogLoss')

ans =

    1.4273

>> rng(42)
>> cost_function(ones(1,4), rand(1,4), 'LogLoss')

ans =

    0.4644

>> rng(42)
>> cost_function(randi(3,3,4), rand(3,4), 'CrossEntropy')

ans =

    7.7579

>> rng(42)
>> apply_activation(randn(3,4), 'softmax', true)

ans =

    0.1039    0.5930    0.0688    0.2363
    0.4237    0.1563    0.6306    0.0615
    0.4724    0.2508    0.3006    0.7022

```

```
>> rng(42)
>> cost_derivative(randi(3,3,4), rand(3,4), 'CrossEntropy', 'softmax')

ans =

   -1.1676   -1.8166   -0.5681   -2.8605
   -2.7877   -0.6958   -2.7088   -0.7079
   -2.8182   -0.4752   -1.3881   -2.6336
```

After that you are now able to fit all the datasets. and fully use the graphical interface. In part 2 we will see some not yet implemented features that could be added.

3 Part 2: Extra features

In this part, we will see some features that could be added to the graphical interface or to the training in general. You are free to select the ones you prefer and implement them. You should at least implement two of them or select only one but studying it extensively. For clarity extra ideas on each subjects are listed in sperate paragraphs.

3.1 Learning rate decay

As stated during the lecture of applied machine learning, a fixed learning rate is problematic as, if set too high it might makes the convergence impossible, while if sets too low makes the training slower. The learning rate could be adapted throughout the training. A common approach is to fix a learning rate decay d where,

$$\eta_{t+1} = d.\eta_t, \tag{8}$$

with $d = 0.99$ for example. Such a technique add an extra parameter that could be set via the graphical interface. It will impact the function `update_parameters.m` in `MyNeuralNetworks.m` class.

To go further

There are many other ways to update the learning rate and change the update rule in general (see here for an overview).

3.2 Input normalization

Usually, we do not perform the learning on the raw data as it impacts the performance. The common approach is to first apply some normalization on the input data. Common normalization is, either to put all the input data between 0 and 1 or to substract the mean and divide by the range. This is referred as `feature scaling` (see `https://en.wikipedia.org/wiki/Feature_scaling` for different methods). One could simply select the possibility to apply normalization in the data via the graphical interface (an example button is already added) or add an extra menu to select which normalization technique to consider.

To go further

Nowadays, most deep learning implementations go even further by adding the possibility to normalize the data at each layers of the network. A technique referred as `batch normalization` (see here for an overview of the method).

3.3 Gradient clipping

If you tried to implement the Leaky ReLU in the activation functions and set a large learning rate you have probably noticed that Matlab throws some errors and the plot are blanks. This comes from the fact that the gradient goes to infinity and, therefore, the output value of the network will be `NaN`. To prevent that, one should implement a method called gradient clipping. The algorithm is pretty simple, if $||\frac{\delta E}{\delta A^l}|| > \epsilon$ with ϵ large than you clip the gradient to a value given by,

$$
\begin{aligned}
\nabla &= \frac{\delta E}{\delta A^l} \\
\nabla &= \frac{\epsilon}{||\nabla||}\nabla.
\end{aligned}
\tag{9}
$$

This should prevent the gradient from exploding. This change would impact the `backpropagation` function as it should be calculated for each layers.

3.4 Regularizations

One common problem of large neural networks is overfitting to the training data as some models might easily reach thousands ans thousands of parameters[5]. If you suspect you neural networks to be overfitting your data, you could try to add `regularization`.

There are many regularization but the most common are $L1$ and $L2$ regularization. Both regularizations consist of adding a term to the cost function, in order to add more sparsity in the network, i.e. ensuring that no neurons take a very large value, overpowering the other ones. It might even associate certain neurons with a zero weights, which will simplify the model[6]. The term to ass to the cost function is therefore a cost based on the weight of the network. Adding a term to the cost function will obviously impact the backpropagation update rule. Let us start with $L1$ regularization,

$$
\begin{aligned}
E(Y^d, Y)_{L1} &= E(Y^d, Y) + \frac{\lambda}{2M}\sum_{l=1}^{L}\sum |W^l| \\
\frac{\delta E_{L1}}{\delta W^l} &= \frac{\delta E}{\delta W^l} + \frac{\lambda}{m} sgn(W^l),
\end{aligned}
\tag{10}
$$

where $\sum |W^l|$ is the sum of the absolute values of all the weights for layer l, i.e. $\sum |W^l| = \sum\limits_{i=1}^{S^{l-1}}\sum\limits_{j=1}^{S^l} |w^l_{ij}|$, and $sgn(W^l)$ is the sign function which returns $+$ for all positive values of W^l and $-$ for the negative ones. $L2$ regularization is very similar,

$$
\begin{aligned}
E(Y^d, Y)_{L2} &= E(Y^d, Y) + \frac{\lambda}{2M}\sum_{l=1}^{L} ||W^l||^2 \\
\frac{\delta E_{L2}}{\delta W^l} &= \frac{\delta E}{\delta W^l} + \frac{\lambda}{m} W^l,
\end{aligned}
\tag{11}
$$

[5]Think about fitting sample points lying around a straight line with a 12 degree polynomial function. A linear regression would be perfectly sufficient and more effective.

[6]Same as putting a 0 value in the weights of a polynomial regression

where $||W^l||^2$ is the Frobenius norm, i.e. $||W^l||^2 = \sum_{i=1}^{S^{l-1}} \sum_{j=1}^{S^l} (w_{ij}^l)^2$. In both regularization techniques, the λ value becomes another `hyperparameter` of the network. A too small value would introduce almost no regularization, while a too large values would generally lead to underfitting.

To go further

There is another commonly used regularization technique called `dropout`. Its aim is similar to both $L1$ and $L2$ as the goal is to avoid some neurons to take too large a value. However the principle is much different as instead of modifying the cost function and the backpropagation, this method impacts only the forward pass during training. The idea is to randomly deactivate some neurons, such that, if the network started relying too much on those, when they get deactivated, the prediction will be completely off leading to a large error (have a look at the original paper for more information on the subject).