Master 1 Computer Science

Memoire

Génération de modèles de réseaux de neurones

Théo Maria
Brian Dietrich
Brian Lebreton
Emmanoe Delar
Najeebullah Ahmadzai

April 4, 2019

Clients : Linh Van Le and Marie Beurton-Aimar
Teacher : Adrien Boussicault

université de BORDEAUX

Contents

Chapter 1

Context

1.1 Summary

With the availability of powerfull hardware as GPUs (Graphics Processing Unit) wich were initialy built for video game and could operate huge mathematical operations quickly, data bank and algorithms, deep learning became very popular. A lot of graphical user interface for training and developping deep neural network are available. Like Expresso for Caffe, NVidia DIGITS for Theano, Caffe, Torch and Tensorflow. Unfortunately they are not compatible with the PyTorch library at the time when we began this project. The alternative we propose is an application that will be an interface for PyTorch for training and developping deep neural network with the Pytorch framework.

1.2 Domain descripton

Our application is developed for user who want to build/modify neural network model with PyTorch, a recent deep learning framework developed by Facebook since October 2016. Deep Learning algorithms are a part of the machine learning family algorithms. Machine learning is one of the fields of study of Artificial Intelligence. It relies on statistical concepts as well as models to give machines the ability to learn. It basically consists of showing a lots of examples, using a set of data, the training set, to make our algorithm try to recognize patterns in it, so it can learn to recognize new data instantly thus come up with rules to predict outcomes for unseen data:

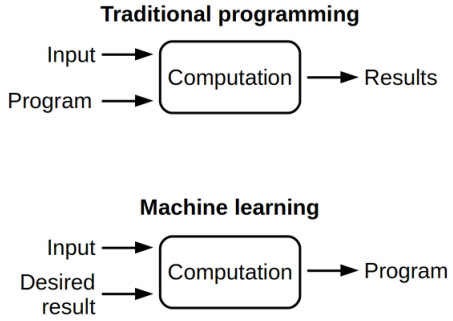


Figure 1.1: Traditional Programming vs Machine Learning

The training phase of a machine learning algorithm takes the training set, and tries for a pre-determined number of iterations, to classify the data. In each iteration, the algorithm will take into account the errors of classification in the last iteration to correct itself, step by step, until its accuracy is high enough.

There are different types of approach for machine learning algorithms :

- Supervised learning : In this case, the data is annotated, which means the algorithm knows the inputs and the outputs of the set. It allows to directly check if the result is correct in each iteration.
- Semi-supervised learning : In this case it's the same as supervised learning, with the exception that only a part of the data is annotated
- Unsupervised learning : This type of approach is clearly distinct from the two others, since the data is not annotated. In this case, the algorithm will try to find patterns by itself, without the use of external corrections. It's only after training that the user can see the results.

Once the training of the algorithm is finished, we can evaluate it by using another data-set, which contains different data than the training set, to evaluate the accuracy of our algorithm. Here is a visualization among AI, ML and DL.

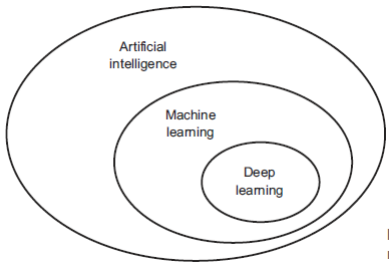


Figure 1.2: About AI

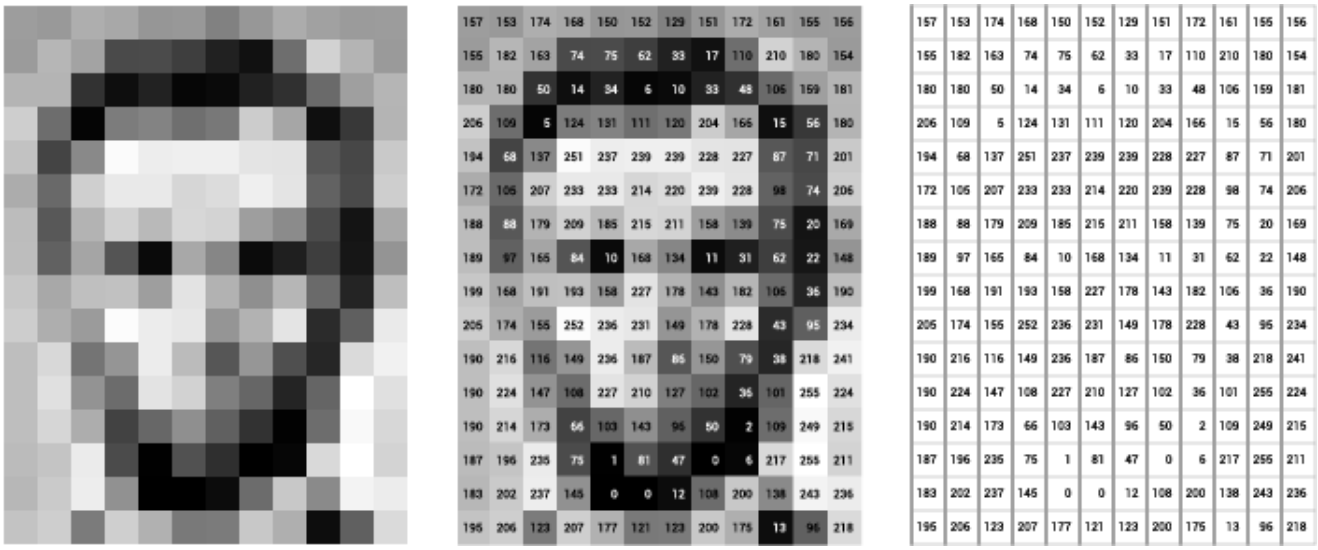
Deep Learning algorithms are a part of the machine learning family algorithms. To be able to recognize patterns in data, they use a cascade of multiple layers of nonlinear processing units. Each layer uses the output from the previous one. They work with the concept of deep neural networks.

1.2.1 How a computer deals with input data

When a computer operates on an image, it will obtain an array of pixel values that the numbers of pixels would depend on the size of image but will have three values or more specifically can say three dimensions. The first and second dimensions are the height and wide of image and the third value will represent the coloring scheme Red, Green, Blue in short RGB. For instance, we have a 480 X 480 image, the represented image will be 480 X 480 X 3.

Each of these numbers, between 1-480 X 480 (according to our example) is given a value from 0 to 255 which describes the pixels' intensity at that point; means if a pixel color is white, it will has a value which corresponds the white color (more likely zero) and if a pixel color is red, it will has a value corresponded to the red color. The intensity of the all different colors are in the range of 0 to 255.

All these values would be in a array for a picture and this is how a picture looks for a computer. Once we give the computer this array of numbers, it will read each pixels and then it will generates probability function that each pixel is more likely to which category for



157	153	174	168	150	152	129	151	172	161	155	156
155	182	163	74	75	62	33	17	110	210	180	154
180	180	50	14	34	6	10	33	48	106	159	181
206	109	5	124	131	111	120	204	166	15	56	180
194	68	137	251	237	239	239	228	227	87	71	201
172	105	207	233	233	214	220	239	228	98	74	206
188	88	179	209	185	215	211	158	139	75	20	169
189	97	165	84	10	168	134	11	31	62	22	148
199	168	191	193	158	227	178	143	182	106	36	190
205	174	155	252	236	231	149	178	228	43	95	234
190	216	116	149	236	187	86	150	79	38	218	241
190	224	147	108	227	210	127	102	36	101	255	224
190	214	173	66	103	143	96	50	2	109	249	215
187	196	235	75	1	81	47	0	6	217	255	211
183	202	237	145	0	0	12	108	200	138	243	236
195	206	123	207	177	121	123	200	175	13	96	218

Figure 1.3: Pixel Demo

example (.80 for cat, .15 for dog, .05 for bird, etc). At the end, each pixel will be classified that whether its more likely a cat, dog or a bird.

The part of the program that decides whether a pixel is more likely belong to a dog, a cat or a bird, is called Perceptron in Deep learning terminology.

In Deep learning, the Perceptron is an algorithm or a part of Artificial Neural Network which decides whether or not an input, represented by a vector of numbers, belongs to some specific class. It is a type of linear classifier, i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector.

1.2.2 Neural network

Neural networks are a set of algorithms, which are designed to recognize patterns. They read and interpret sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors and arrays of three dimensions.

A neural network contains layers of interconnected nodes. Each node is a perceptron and is similar to a multiple linear regression. The perceptron feeds the signal which is produced by a multiple linear regression into an activation function that may be nonlinear.

Neural networks help us cluster and classify. Neural Network cam be considered as a clustering and classification layer on top of the raw data stored. They help to group unlabeled data according to similarities among the example inputs and they classify data when they have a labeled dataset to train on. Neural networks can also extract features that are fed to other algorithms for clustering and classification; so deep neural networks can be considered as components of larger machine-learning applications involving algorithms for reinforcement learning classification and regression.

1.2.3 Layer

A layer is a collection of 'nodes' operating together at a specific depth in a neural network. Layers can be defined into three types:

- **Input layer**: Contains the raw data or more specifically datasets.

- **Hidden layer(s)**: each layer tries to learn different aspects about the data by minimizing an error/cost function.
- **Output layer**: Is a simple node, which contains the result.

1.2.4 Nodes

A neural network contains layers of interconnected nodes. Each node is a perceptron and is similar to a multiple linear regression. The perceptron feeds the signal which is produced by a multiple linear regression into an activation function that may be nonlinear.

1.2.5 Optimizer function

Optimization generally is the process of or the selection of a best element (with regard to some criterion) from some set of available alternatives.

Optimization algorithms helps us to minimize or maximize an Objective function which is simply a mathematical function dependent on the Model's internal learnable parameters which are used in computing the target values(Y) from the set of predictors(X) used in the model. For example we call the Weights(W) and the Bias(b) values of the neural network as its internal learnable parameters which are used in computing the output values and are learned and updated in the direction of optimal solution i.e minimizing the Loss by the network's training process and also play a major role in the training process of the Neural Network Model.
Optimization Algorithm falls in 2 major categories

1. **First Order Optimization Algorithms**
   These algorithms minimize or maximize a Loss function using its Gradient values with respect to the parameters. Most widely used First order optimization algorithm is Gradient Descent.The First order derivative tells us whether the function is decreasing or increasing at a particular point.
2. **Second Order Optimization Algorithms**
   Second-order methods use the second order derivative which is also called Hessian to minimize or maximize the Loss function.

1.2.6 Loss function

As we have mentioned above, we must learn our network by showing the inputs and the associated output. Then our network compute a model and by give this model unseen input, it predict an output and we tell it if the prediction is correct or not. It then correct its mistake until it reach a certain level of accuracy fixed by the user. This is when loss functions appear.

1.2.7 Linear Classification

In deep learning(machine learning too) we use linear classification to classify and differentiate a bunch or a group of given data into two part by using their characteristics.

1.2.8 Train a network

The training is one of the most important step for the neural network, it will learn by the input and the outputs that we give to him and update its internal state accordingly, to have the calculated outputs as close as possible from the desired outputs.

For instance we take a simple example to for better comprehension. There are just one input and one output.

Input	Desired output
0	0
1	2
2	4
3	6
4	8

The Network Training process is divided into the following steps:

1. **Model initialization**
   The first step is to start randomly from any where to start the model. Thus a random initialization of the model is required.
   For our numerical case study, let's consider the following random initialization: (Model 1): y=3.x. The number 3 here is generated at random. Another random initialization can be: (Model 2): y=5.x, or (Model 3): y=0,5.x.

2. **Forward propagate**
   The natural step to do after initializing the model at random, is to check its performance. We start from the input we have, we pass them through the network layer and calculate the actual output of the model straightforwardly.

Input	Actual output of model 1 (y = 3.x)
0	0
1	3
2	6
3	9
4	12

3. **Loss function**
   Basically it is a performance metric on how well the NN manages to reach its goal of generating outputs as close as possible to the desired values. At this stage, in one hand, we have the actual output of the randomly initialised neural network. On the other hand, we have the desired output we would like the network to learn. Let's put them all in the same table.
   As we can see that Actual output is closer to Desired output.

4. **Differentiation**
   In this step we use different optimization technique for example greedy search, genetic algorithm etc to compare which on result is more closer to the desire output.

Input	Actual output	Desired output	Absolute Error
0	0	0	0
1	3	2	1
2	6	4	2
3	9	6	3
4	12	8	4
Total	-	-	10

Input	Output	W=3	rmse (3)	W=3.0001	rsme
0	0	0	0	0	0
1	2	3	1	3.0001	1.0002
2	4	6	4	6.0002	4.0008
3	6	9	9	9.0003	9.0018
4	8	12	16	12.0004	16.0032
Total	-	-	30	-	30.006

5. **Back-propagation**

   Is a method used in artificial neural networks to calculate a gradient that is needed in the calculation of the weights to be used in the network. BackPropagation is shorthand for "the backward propagation of errors," since an error is computed at the output and distributed backwards throughout the network's layers. It is commonly used to train deep neural networks.

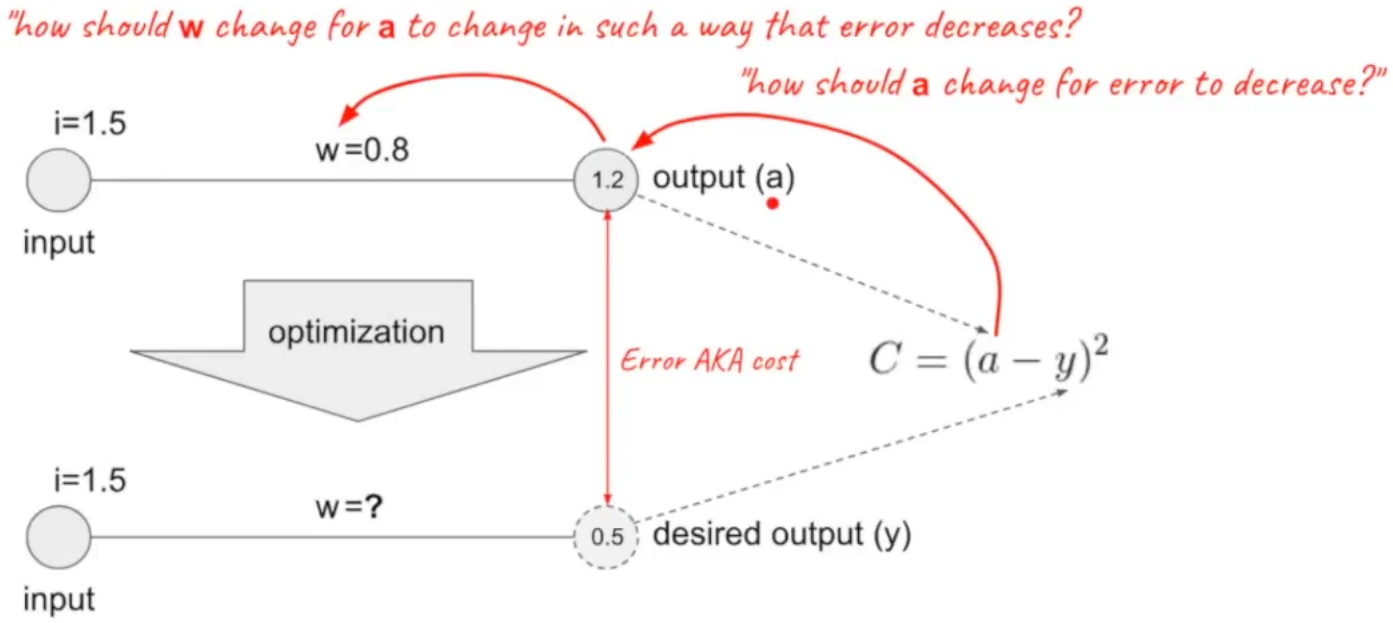


Figure 1.4: Back Propagation

So the Back-propagation process is "how should **w** change for **a** to change in such a way that error decreases"?

6. **Weight update** Weight Update is the process to change the values of weight according to value received from optimization function in order to minimize the cost.

7. **Iterate until convergence** Since we update the weights, it will take several iterations in order to learn and reach to the almost Desired value.

   How many iterations are needed to converge?

   This depends on how strong the learning rate we are applying. High learning rate means faster learning, but with higher chance of instability.
   It depends as well on the meta-parameters of the network (how many layers, how complex the non-linear functions are). The more it has variables the more it takes time to converge, but the higher precision it can reach.
   It depends on the optimization method use, some weight updates rule are proven to be faster than others.

8. **Overall picture**

1.2.9 Evaluate a network

The network evaluation consist to give to the network inputs, so he can generate using its internal state the most likely output according to its "training experience".

1.3 Existing analysis

We analyzed the existing projects, resources and their interrelated materials.

1.3.1 PyTorch

PyTorch is an optimized tensor library for deep learning using GPUs and CPUs. In Pytorch, data are represented as tensor or variable.

Tensor

Tensor are Python's numpy array but they can be used with GPU for better performance. In our project we will not use the GPU but the CPU, we still working on this feature it will be avaible in newer version of the app. But tensor still useable with CPUs.
Like arrays, tensors contains elements and can be multidimensional. Frequenntly used tensor are :

- Scalar (0-D tensors)
- Vector (1-D tensors)
- Matrix (2-D tensors)
- 3-D tensors
- Slicing - tensors
- 4-D tensors
- 5-D tensors

torch.rand(10) returns a tensor filled with 10 random numbers from a uniform distribution on the interval [0, 1)

Image can be represented as 3D-Tensor (height,weight, channel (RGB)) so a 4D-Tensor can be a Tensor with a batch of images. The number of images are represented in the 4th dimension. Slicing tensor are one-dimensional tensor that we slice the elements. For example 1-D_tensor*[:slice_index]*

Variables

Pytorch variables can be seen as containers with:

- Data : tensor object
- Gradients : rate of the change of the loss function
- Creator: reference to the function that created it

Here is an example on how to create a variable containing a 2x2 tensor:

```
>> x = Variable(torch.ones(2,2),requires_grad=True)
>> x.data
tensor([[1.,  1.],
        [1.,  1.]])
>> x.grad
tensor([[0.2500,  0.2500],
    [0.2500,  0.2500]])
```

To compute the actual gradient we must apply the mean() first then the backward() function on x.

1.3.2 Computation graphs

To represent deep learning algorithms we can use graphs. For a linear relationship model represented as

$$y = wx + b$$

The associated computation graph and its implementation in PyTorch can be:

Computation graph

Here, each circle represents a variable. Squared symboles reprensents operators.

PyTorch Network implementation

```
def linear_model(x):
    y = torch.matmul(x,w)
    return y
```

1.3.3 Layer

Torch.nn

Torch.nn is a package containing mathematical operations and provides many utilities for efficient seralizing of Tensors. To build a working deep neural network we must set a collection of 'nodes' operating together in the neural network. Pytorch provides a higher level abstraction in torch.nn called layers wich create layer much simpler.

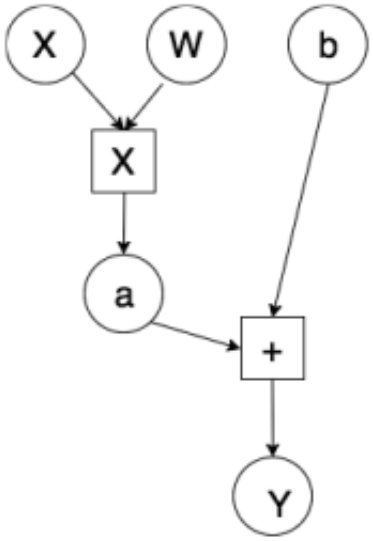


Figure 1.5: Linear relationship graph

```
CLASS torch.nn.Linear(in_features, out_features, bias=True)
```

Applies a linear transformation to the incoming data: y=Ax+b

Parameters:

- in_features – size of each input sample
- out_features – size of each output sample
- bias – If set to False, the layer will not learn an additive bias. Default: True

The previous model can be represented as a torch.nn layer as follow:

```
f = nn.Linear(17,1)
```

Here a graphical representation of a Convolutional Neural Network

Convolution layers

For an input image as 32x32x3 (3 color channels) if we want to apply a convolution with a filter 5x5x3 (kernel size) we would apply the torch.nn.Conv2d() function

```
CLASS torch.nn.Conv2d(in_channels, out_channels, kernel_size)
```

More convolution functions in torch.nn:

- Conv1d
- Conv2d
- Conv3d
- ConvTranspose1d
- ConvTranspose2d
- ConvTranspose3d
- Unfold
- Fold

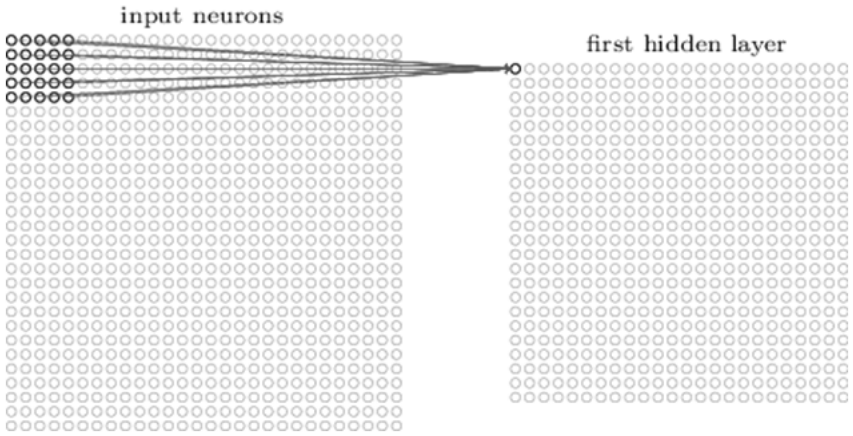


Figure 1.6: Visualisation of 5x5 filter producing an activation map

Pooling layers

In a CNN, max-pooling, is the most common type of pooling layer used as the next step after convolutional layer. The convolutional layer pass it an input and its work is to reduce the dimensionality of this input. For images, it reduce the number of pixel from the previous convulutional layer. Main goal of this technic is:

- By having less spatial information you gain computation performance
- Less spatial information also means less parameters, so less chance to over-fit
- You get some translation invariance

All this technical informations are abstracted in PyTorch. If we want to set an pooling layer we just have to pay attention to some parameters:

- Input: H1 x W1 x Depth_In x N
- Stride: Scalar that control the amount of pixels that the window slide.
- K: Kernel size

PyTorch abstraction :

```
CLASS torch.nn.MaxPool2d(kernel_size, stride=None)
```

All the pooling function in pytorch:

- MaxPool1d
- MaxPool2d
- MaxPool3d
- MaxUnpool1d
- MaxUnpool2d
- MaxUnpool3d
- AvgPool1d
- AvgPool2d

- AvgPool3d
- FractionalMaxPool2d
- LPPool1d
- LPPool2d
- AdaptiveMaxPool1d
- AdaptiveMaxPool2d
- AdaptiveMaxPool3d
- AdaptiveAvgPool1d
- AdaptiveAvgPool2d
- AdaptiveAvgPool3d

If user want to use theses functions, they will be listed in a scroll bar in the network edit layer mode.

1.3.4 Dropout

Dropout refers to ignoring neurons during the training phase of certain set of neurons which is chosen at random, the ignored neurons will not be considered during a particular forward or backward pass.
At each training stage, individual nodes are either dropped out of the net with probability 1-p or kept with probability p, a smaller network is "created", incoming and outgoing edges to a dropped-out node are removed.
List of all **Dropout functions**:

- dropout
- alpha_dropout
- dropout2d
- dropout3d

1.3.5 FC Linear

1.3.6 Loss function

As we say upper, we must learn our network by showing the inputs and the associated output. Then our network compute a model and by give this model unseen input, it predict an output and we tell it if the prediction is correct or not. It then correct its mistake until it reach a certain level of accuracy fixed by the user. This is when loss fuctions appear. Pytorch provide them in torch.nn, As example, the Mean Absolute Error measures the numerical distance between the estimated and actual value.

```
CLASS torch.nn.L1Loss(size_average=None, reduce=None, reduction='mean')
```

Parameters:

- size_average (bool, optional) – Deprecated (see reduction). By default, the losses are averaged over each loss element in the batch. Note that for some losses, there multiple elements per sample. If the field size_average is set to False, the losses are instead summed for each minibatch. Ignored when reduce is False. Default: True
- reduce (bool, optional) – Deprecated (see reduction). By default, the losses are averaged or summed over observations for each minibatch depending on size_average. When reduce is False, returns a loss per batch element instead and ignores size_average. Default: True
- reduction (string, optional) – Specifies the reduction to apply to the output: 'none' — 'mean' — 'sum'. 'none': no reduction will be applied, 'mean': the sum of the output will be divided by the number of elements in the output, "sum": the output will be summed. Note: size_average and reduce are in the process of being deprecated, and in the meantime, specifying either of those two args will override reduction. Default: "mean"

Here is the list of all the Loss Functions provided by PyTorch:

- L1Loss
- MSELoss
- CrossEntropyLoss
- CTCLoss
- NLLLoss
- PoissonNLLLoss
- KLDivLoss
- BCELoss
- BCEWithLogitsLoss
- MarginRankingLoss
- HingeEmbeddingLoss
- MultiLabelMarginLoss
- SmoothL1Loss
- SoftMarginLoss
- MultiLabelSoftMarginLoss
- CosineEmbeddingLoss
- MultiMarginLoss
- TripletMarginLoss

1.3.7 Optimizer

Optimizer's goal is to tie together the loss function and model parameters by updating the model into its most accurate possible form by changing the nodes' weight in response to the output of the loss function.
Here is the list of all the Optimizer Functions provided by Pytorch:

- Adadelta
- Adagrad
- Adam
- SparseAdam
- Adamax
- ASGD
- LBFGS
- RMSprop
- Rprop
- SGD

Caffe and Expresso

To develop our application we think about looking for a existing graphic interface for a deep learning library, Caffe, a deep learning framework already have a graphic interface Expresso but today Caffe's architecture is average.
Because of the commonalities between Pytorch and Caffe we have decided to inspired our application with Expresso.

Caffe

Caffe is a deep learning framework developed by Berkeley AI Research,got an excellent convolutional network implementation used for image processing like image classification, face recognition. Unfortunately it's too cumbersome if we have a big networks compose of many layers.

Expresso

Expresso has been developed, to graphically control Caffe library. The framework itself is written in C++, but it has a python interface.
Expresso is a Python-based graphical user interface made for designing, training, and exploring deep-learning frameworks. It is built atop of Caffe. Its main purpose is to facilitate the use of Caffe by providing a user interface which allow to use most of its features graphically.

Expresso has a handy importing tool which enables users to import different formats of data such as Text, LevelDB, .mat, HDF5 and Folder formats. It provides additional information regarding to the data being imported in a detailed manner. A technical browser tool is provided, which can be used for a quick preview of the imported data. The imported data can be exported to the other formats.

Another interesting feature is that the back-end framework related to Data view also features a parser. This parser can be utilized to quickly extend the import interface to load data in formats not provided yet in Data view.
Expresso interface is separated into four different views : Data View, Net View, Train View and Exp View These different views will basically be our main inspiration for how we organize our application.

- Data view allows to import data, by selecting the format of the file. In addition, it provides support for basic data manipulation. The back-end framework of data view also allows to program a new import functionality, which allows to import formats that are not supported yet.

- Network view is used to design a deep network from scratch. Each layer of the network can be created and modified using an edit interface, which automatically provides contextual layer description, as the user types. They are color-coded by type, for easy visualization. The "train net" tab allows to create a net, and the "deploy net" tab allows to modify an existing one. All the created networks are Caffe-compatible, which means the user can use them outside of Expresso as well.

- The train view is used to train the deep networks. Expresso allows training to be stopped midway, and resumed. In addition to conventional deep network training, it also allows provides support for training external classifiers. These classifiers are trained on features obtained by passing data through trained net.

- Experiment view can be used for three important tasks :

  - Feature extraction using featuring nets : In the deep-learning field, a "feature" is basically a pattern that can be found in a set of data. For example, it can be used to distinguish shapes in pictures. Feature extraction is a process that consists of finding the features.
  - Visualizing feature data : This basically allows to visualise the features found in an image, with the use of colors
  - Testing pre-trained models : Once a model is trained with the use of a training data-set, it can be tested with another data-set. By this process, we can determine the accuracy of a model, which will allow to compare it to other models.

The Expresso application is multi-threaded, which means it can enable concurrent execution of tasks. It provides a notification system, which can alert the user when there is an event, or when a task is completed.

Chapter 2

Requirements Analysis

Our clients desired to have a graphic interface like Expresso (working on Caffe framework) for Pytorch. Their requirement is such an interface that the user can navigate between different tab and each tab correspond to one step of building a neural network so that the user be able to train and evaluate the imported data.

2.1 Functional design

We decided to create a user friendly application according to the desire of the client so that its easy for them to understand what they are going to do in each tab and we expressed the functional needs accordingly.

2.1.1 Data tab

- Import Data from the computer (file or directory)that Pytorch can use
- Set the import data as Training Data or Test Data
- Visualize the Data-parameters (Name, Size, Datasets Mode)
- Clear the import data.

2.1.2 Network tab

- Create a network from scratch and edit it.
- Load a network from a specific format (json file) and to visualize it.
- Add a Layer from a database (load in background) to the network.
- Add an Activation Function from a database (load in background) to the network.
- Edit the layer parameter (rename a layer, number edition ...).
- Save a network into a specific format(json file).
- Choose a Loss Function from a database (load in background) and edit its parameters.
- Choose an Optimizer Function from a database (load in background) and edit its parameters.

2.1.3 Train tab

- Choose all Training Parameters (Data to use, Epoch, batchSize).
- Visualize Training result (graphics).
- Train the network.

2.1.4 Evaluation tab

- Choose which data to use for the Evaluation
- Choose a Loss Function and its parameters.
- Evaluate the current network.
- Visualize the network evaluation results.

2.1.5 Use cases diagrams

PyTorchGUI

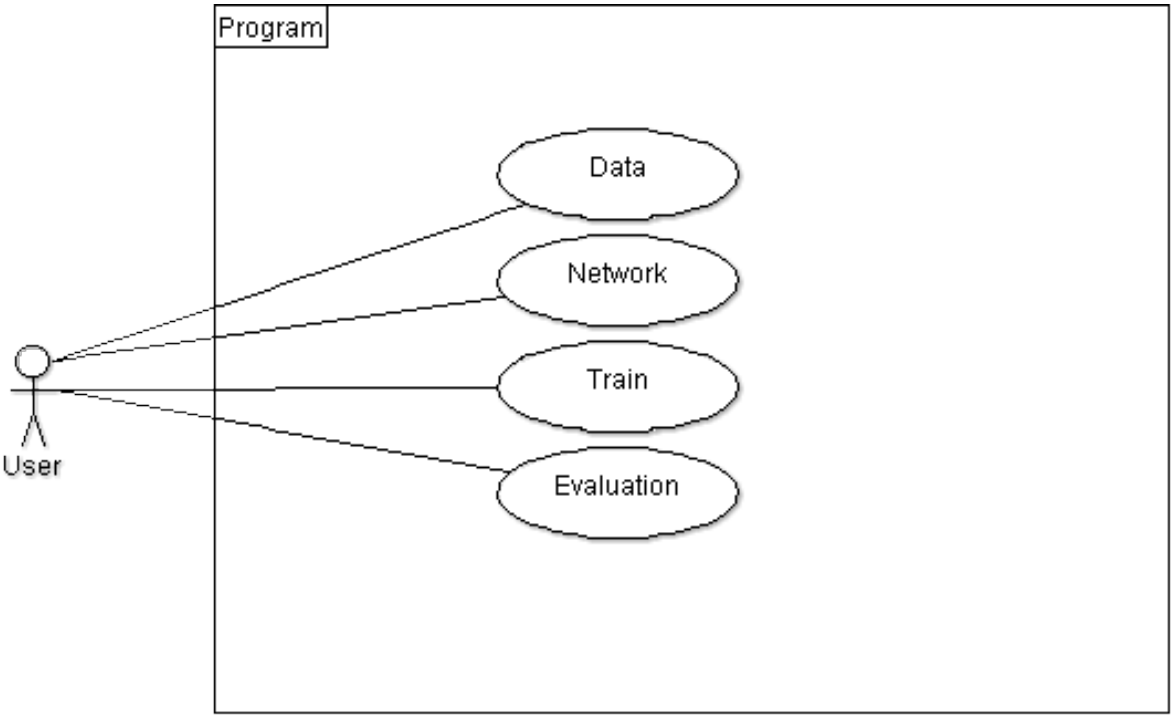


Figure 2.1: Use case PyTorchGUI

There is only one actor that interacts with our application, the user. There is a lot of use cases in our application, it couldn't be possible to put all of them in only one diagram, thus we designed 5 diagrams, one for each tab and one for the program.
From everywhere on the application you can switch of tab:

- Data
- Network

- Train
- Evaluation

Data tab

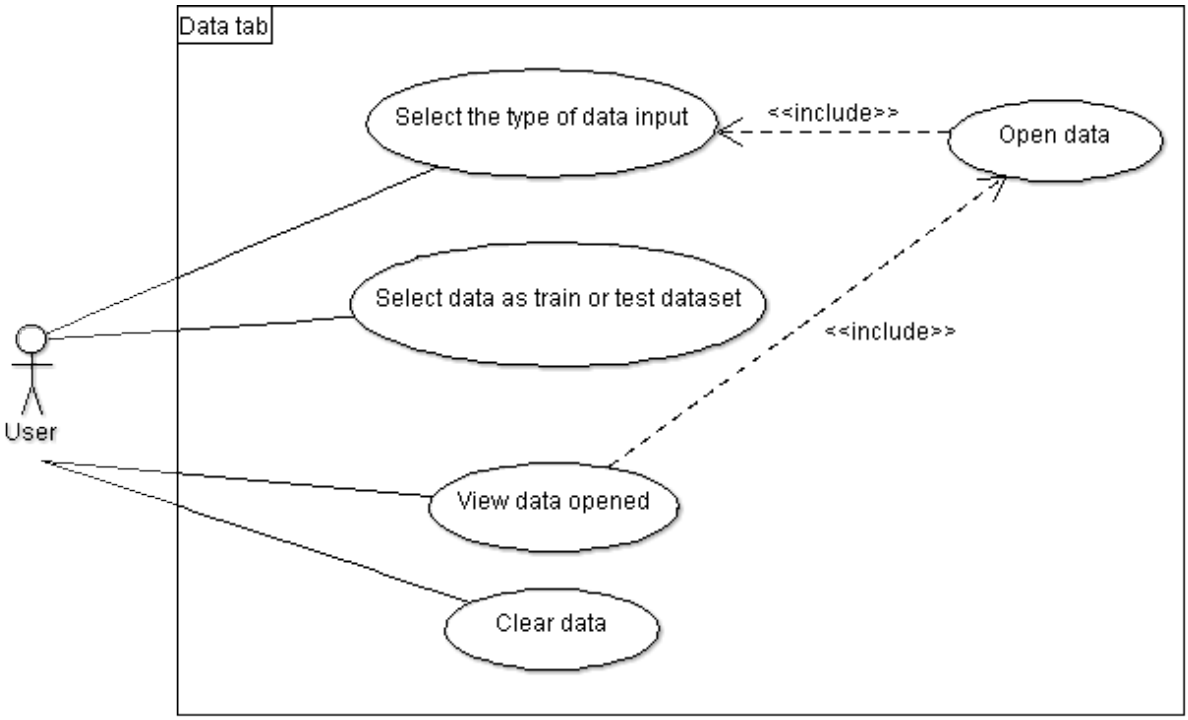


Figure 2.2: Use case Data tab

Network tab

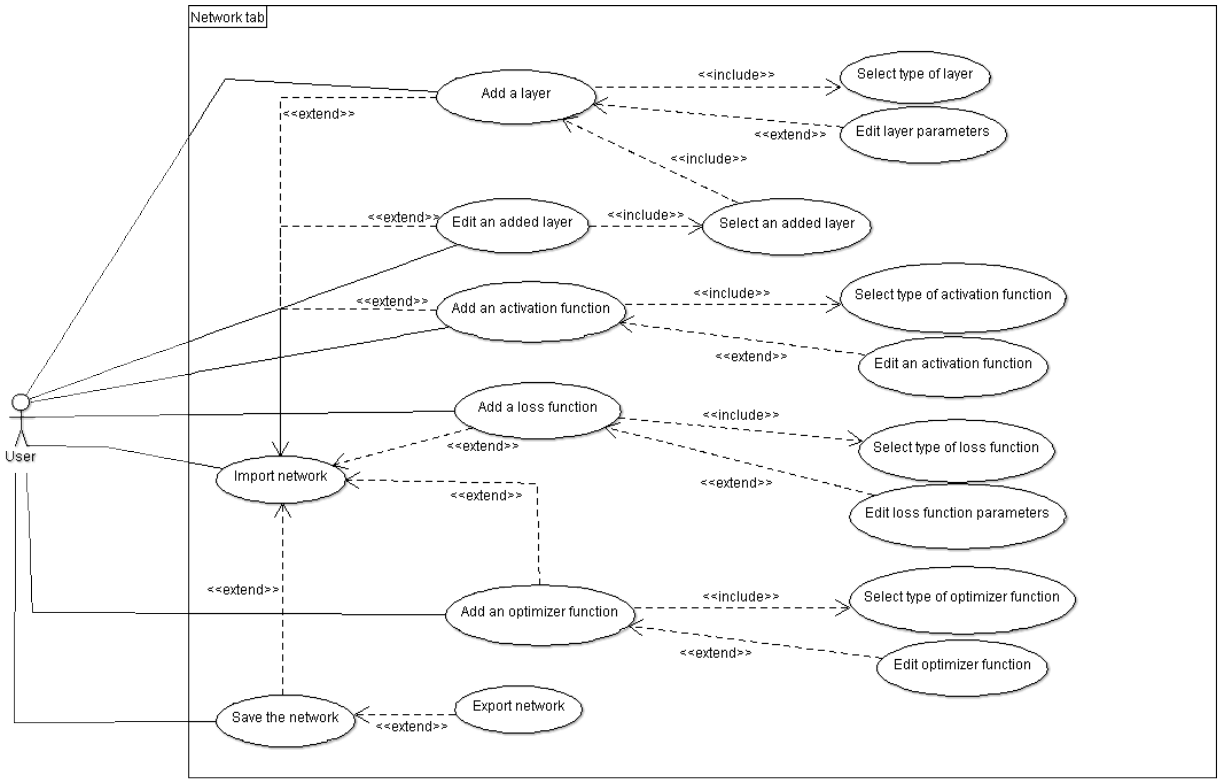


Figure 2.3: Use case Network tab

Train tab

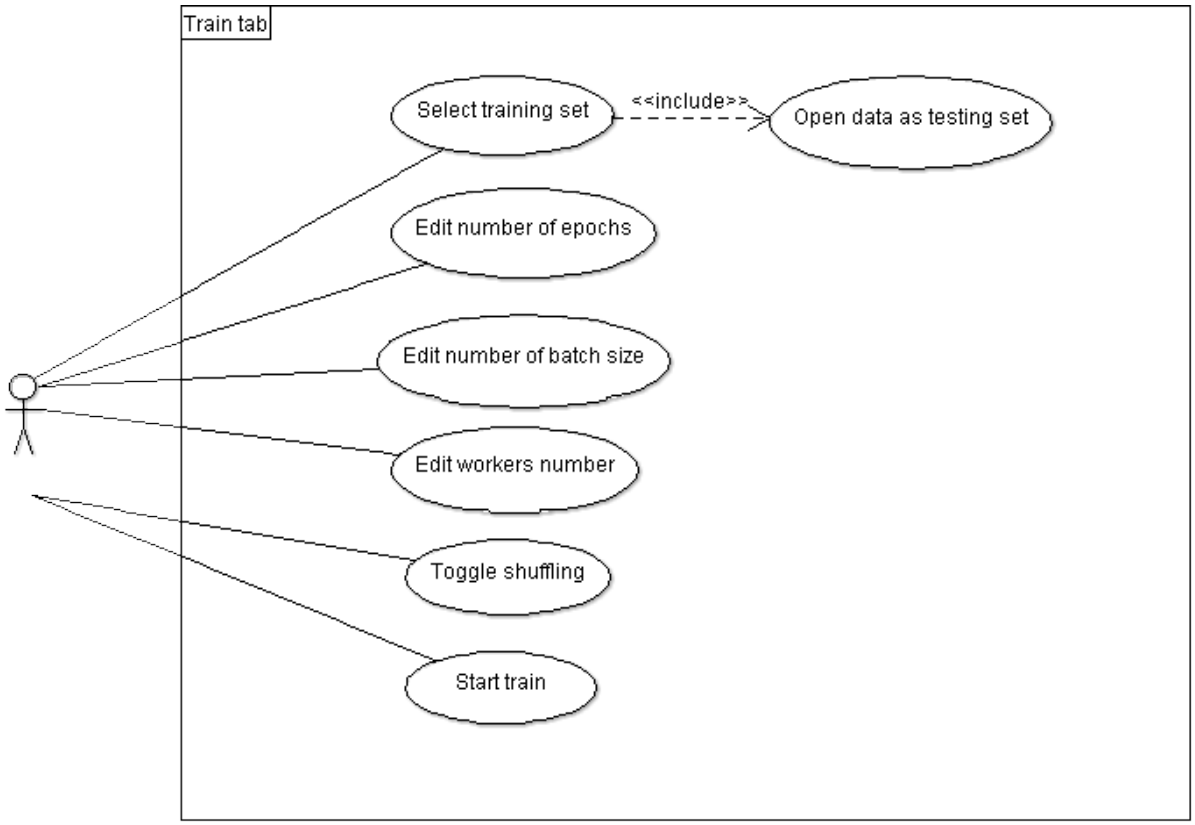


Figure 2.4: Use case Train tab

Evaluation tab

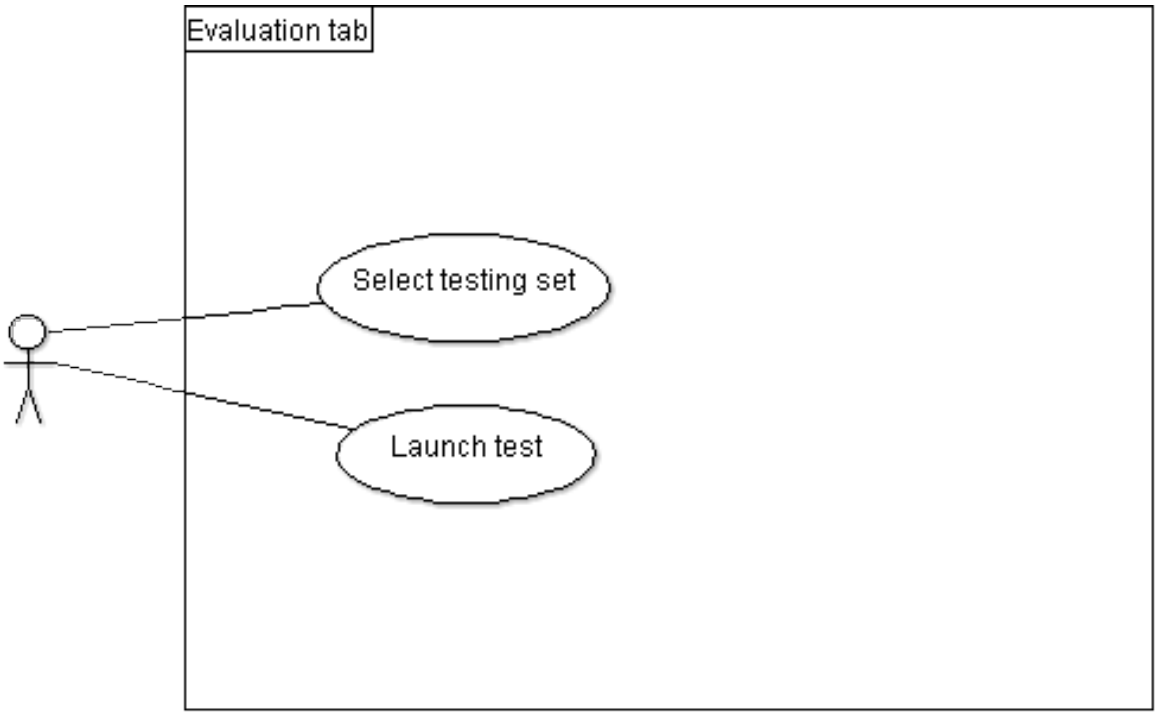


Figure 2.5: Use case Evaluation tab

2.1.6 Scenario

Use case: Build a network from scratch, save it, train it, evaluates it
Actor: User
Summary: The user wants to create his own network that will operate on data he imported. He wants to save it, train his network on a training dataset based on data he imported too and finally evaluate his network.
Precondition:

- The user is using a Unix operating system.
- The user has data that can be imported.

Post-conditions:

- A network has been saved on path user chose.
- A result of the evaluation appeared (whether it is right or wrong).

1. First the user wants to import his data.
2. He selects as which type of data he wants his data imported as.
3. He imports his data by clicking on "Open data".
4. The application opens an explorer window where he select his data.

5. Then the data are sent to PyTorch to be transformed to a dataset.
6. PyTorch returns the dataset to the application.
7. PyTorchGUI display opened data.
8. The user switch to the Network tab.
9. He selects a type of Layer in the dropdown list.
10. Adds it to the network by clicking on "Add a layer".
11. The application opens an edition window.
12. He edits the layer parameters.
13. Clicks on Add.
14. Set it as first layer.
15. Selects a loss function in the dropdown list.
16. Confirm his choice.
17. The application opens an edition window.
18. The user edits the loss function parameters.
19. Clicks on Set.
20. Select an optimizer function in the dropdown list.
21. Confirm his choice.
22. The application opens an edition window.
23. The user edits the optimizer function parameters
24. Save his network
25. The application opens an explorer window.
26. The user choose a path.
27. The application save his network to said path.
28. The user switch to the Train tab.
29. He edits the train parameters.
30. Clicks on "Start Train".
31. The application build the network from the model the user chose previously.
32. Then will train the network thanks to PyTorch.
33. Returns the trained network.
34. The user switch to the Evaluation tab.

35. He selects the dataset to evaluate.
36. Then selects a loss function.
37. Confirm his choice.
38. PyTorchGUI opens an edition window.
39. The user edits loss function parameters.
40. Adds it to the network.
41. The application adds the loss function to the network.
42. The user evaluate the model by clicking on "Launch test".
43. PyTorchGUI evaluate the dataset with created network thanks to PyTorch.
44. The application returns the neural network.
45. Then displays it.
46. The user can see his neural network he created, trained and evaluated.

Sequence diagram

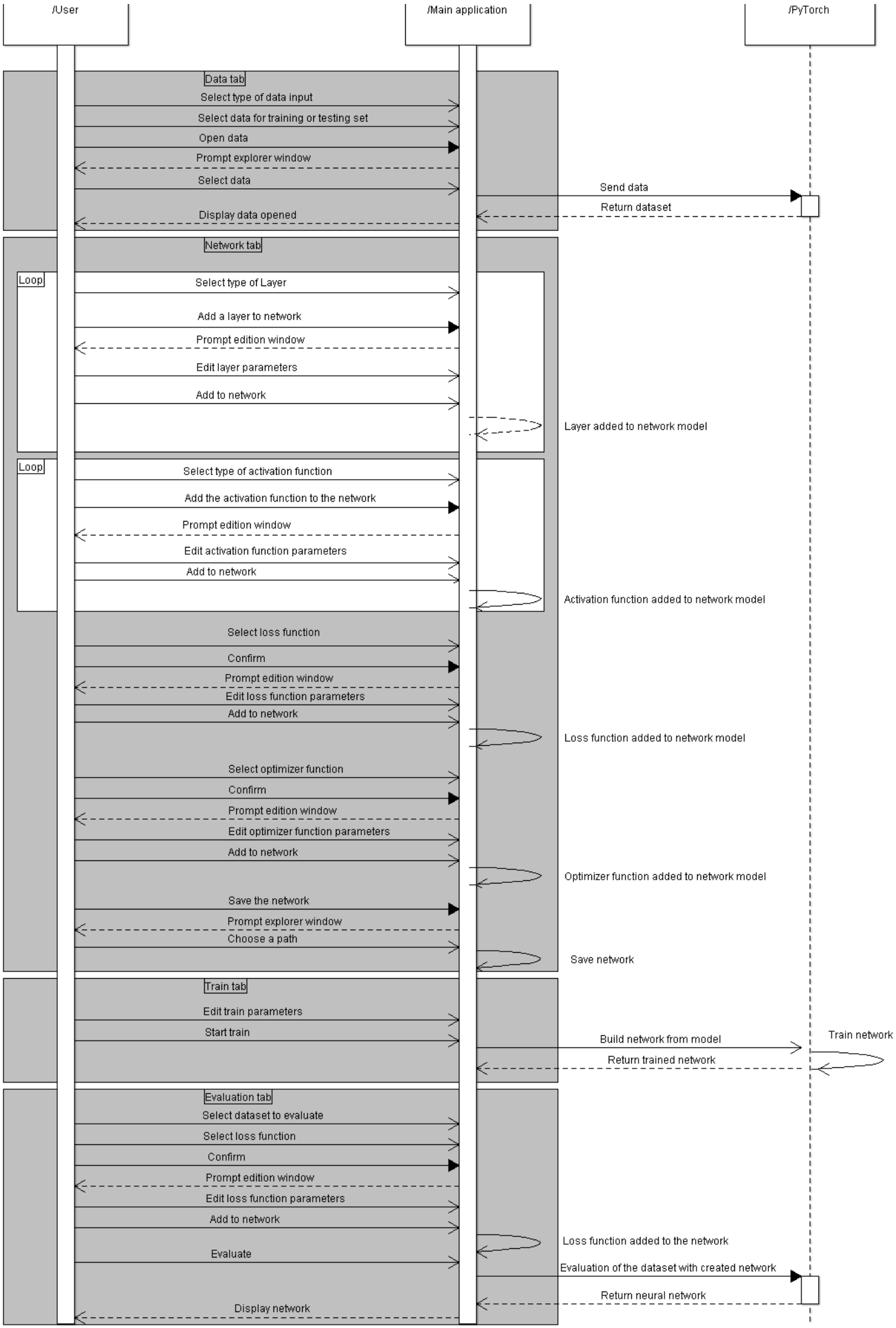


Figure 2.6: Sequence diagram of building network from scratch

2.1.7 Constraints

- Platform-compatibility : The app has to be compatible with Linux. If possible, we will try to make it work also with other platforms.
- Maintainability : We have to develop our application in a way that allows future developers to extend the functionalities. Also, we have to make sure that our app will easily be compatible with future versions of PyTorch.

2.1.8 Choice of application type

To develop our application we were thinking about two types of applications. These two choices are a Web-Service, or a regular application. In order to choose which type of application to develop we drew up a table of advantages and disadvantages.

- Web-Service advantages :
  - The Web application can be used by all users anywhere and when they want if they are connected to Internet. Furthermore, this facilitates a simple to get a multi-platform application.
  - The application is always up to date, if it needs an update we only have to update the server whereas a simple application have to be updated on all users' computer. The app is more flexible.
  - The deployment of the app is easier
  - It could facilitate the extension of the app to add features
- Web-Service drawbacks :
  - Users need to be connected and to have good network performance (fast and stable) especially in case of big data exchanges. In addition, the server could have to support many users at the same time. Even if the app is running in local, they need to have a good central server, which needs a more complex architecture.
  - With the same hardware, performances are lower than a native application, we have to exchange data with the server so it must take a long time before getting results especially if we have a lot of data in addition of the time of calculation.
  - Complex visual effects like drawing a graph could be more complicated to implement
- Native Application advantages :
  - Users don't need an Internet connection.
  - It has got better performances, better time of calculation, better time of response.(Faster than a Web-Service)
- Native Application disadvantages :
  - Every user have to install the application on his personal computer.
  - If the application needs an update, we have to upgrade all the computers equipped with the application.

Our choice : Since both options seemed quite equivalent for us, and since having a web-service deployment wasn't a need for the project, we decided to go on to the **Native Application**. Indeed, this will perfectly allow us to implement all needed features with performance similar to those of PyTorch by itself.

Chapter 3

Architecture and Description of the Application

3.0.1 Architecture of the Application

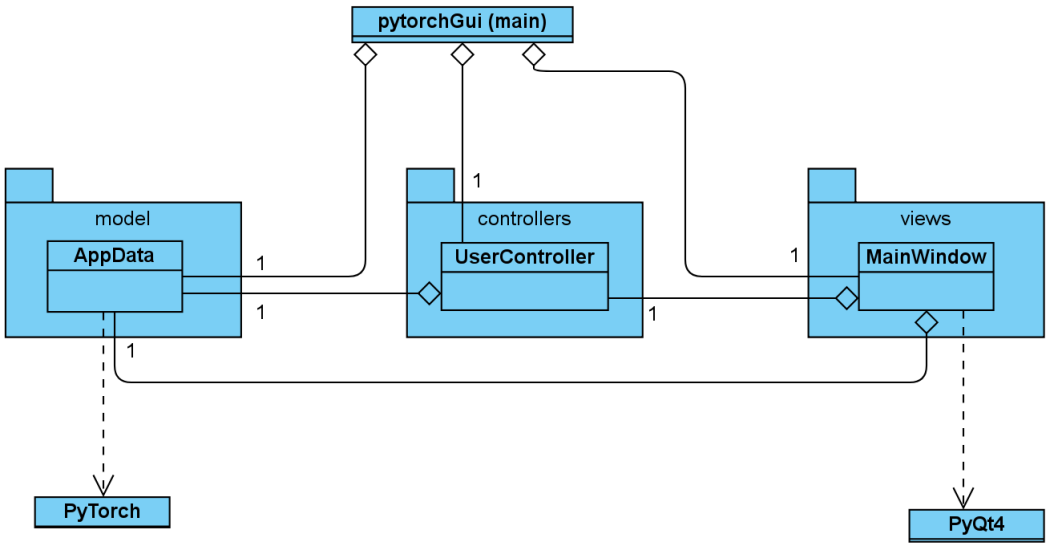


Figure 3.1: Global architecture

The MVC pattern

Before we started to develop our application, we first had to choose a pattern to implement our functionalities. Since we are developing a graphical interface, we decided to implement one of the most commonly used pattern for this kind of application : **the MVC pattern**.
MVC stands for **Model-View-Controller**. It means that the application is separated into three main packages :

- The model : It contains all the logic and data of the application. It is independent from the interface, and we should be able to execute all of the program functionalities from

here.

- The view : It contains the code of the graphical interface. There are widgets and modules used to pilot the application, and others to display results.
- The controller : It is responsible to do the link between the view and the model. It is called by the view when the user requires an action, it validates and transforms the data to make it fit to the model.

Important note : Our application does not strictly follow the MVC pattern. As a matter of fact, in a true MVC pattern, the view should not call the model directly, and should always go through the controller for every action. We found that it was quite heavy to implement in many cases. For example when the view needs a single value from the model, we found it was not necessary to call the controller. Thus, we also added a link between the view and the model, so that the view can call the model directly in those cases.

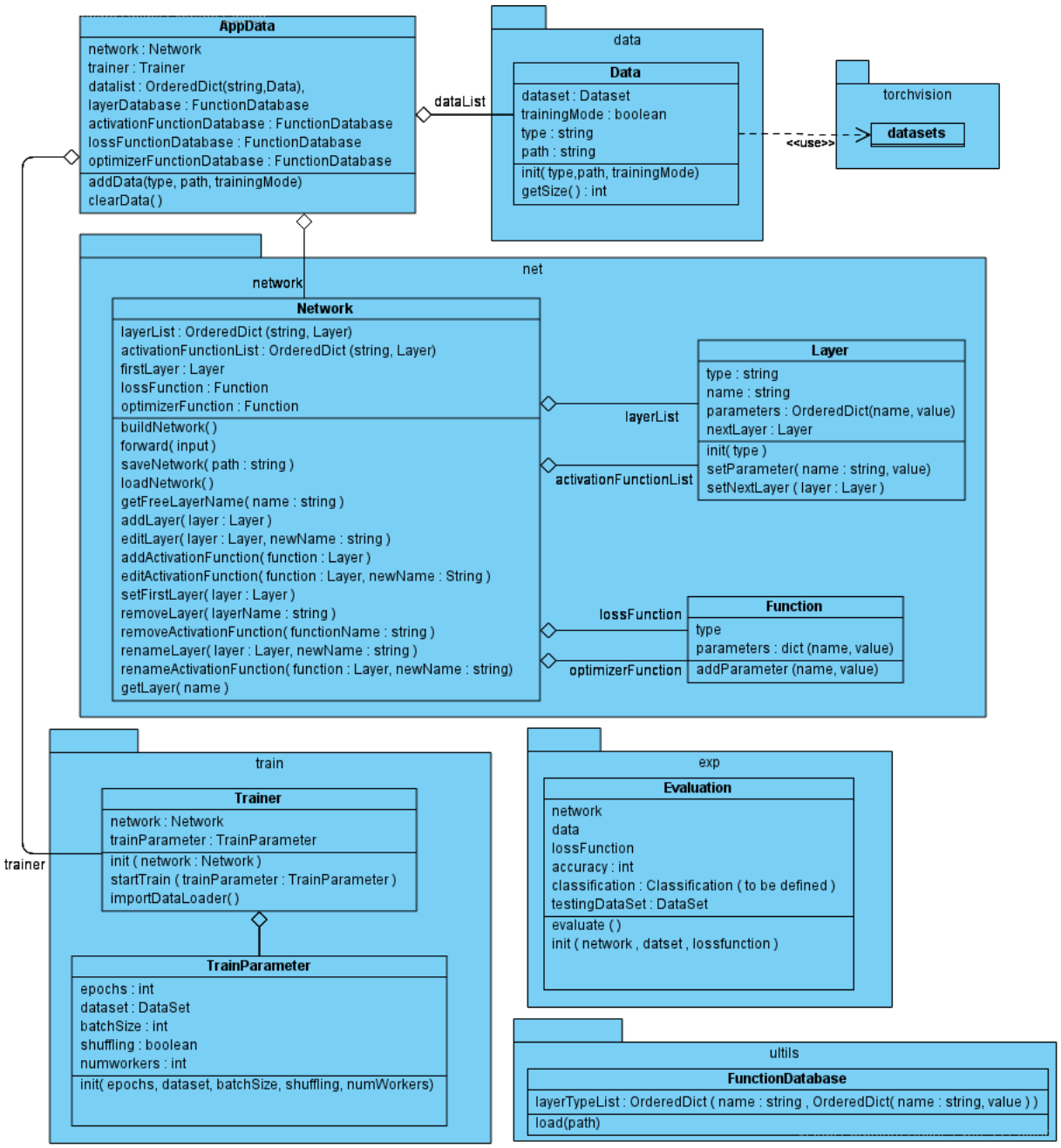


Figure 3.2: Model class diagram

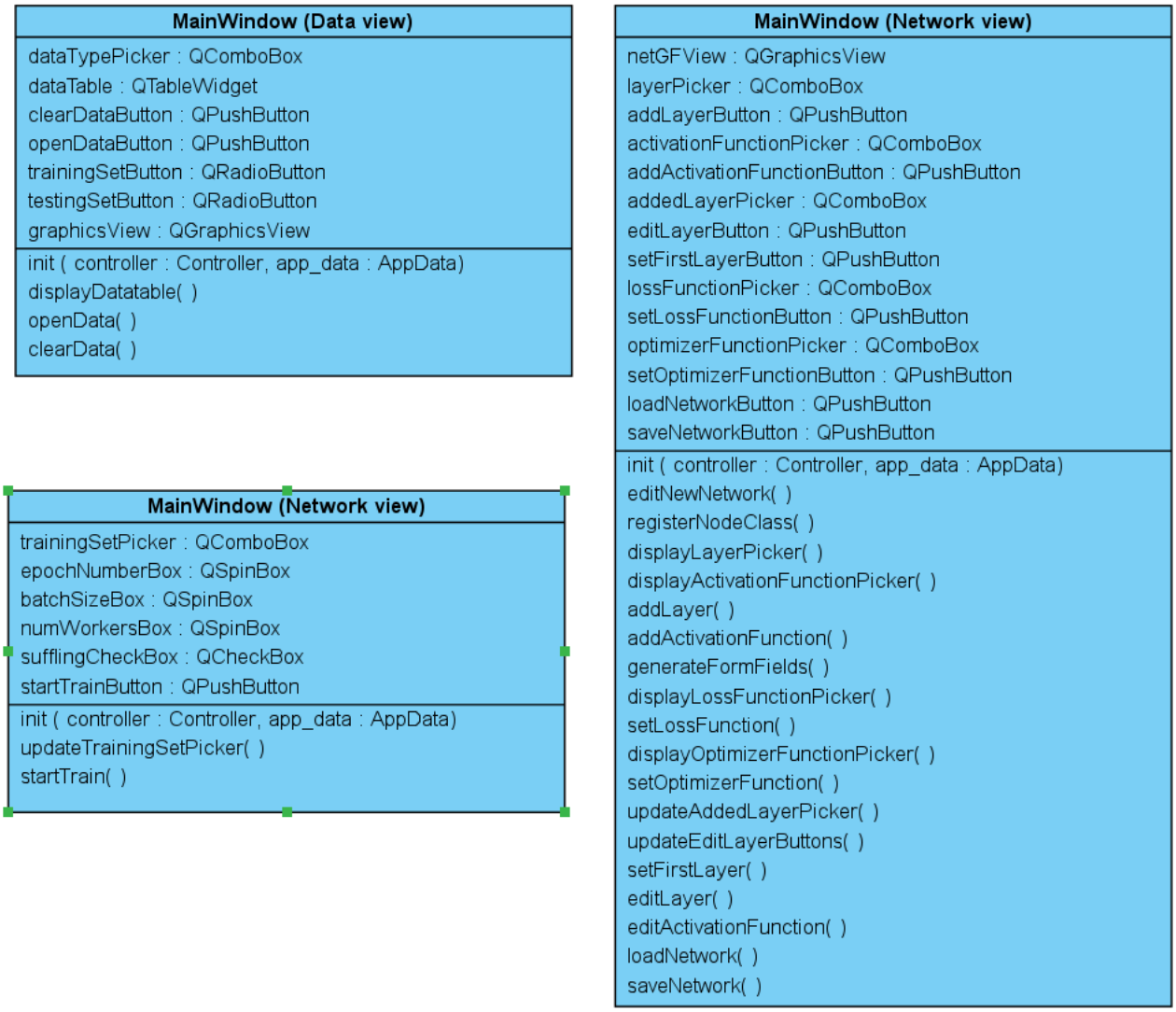


Figure 3.3: View class diagram
These are all **in the same class** (MainWindow), we just separated its content to clarify our explanation

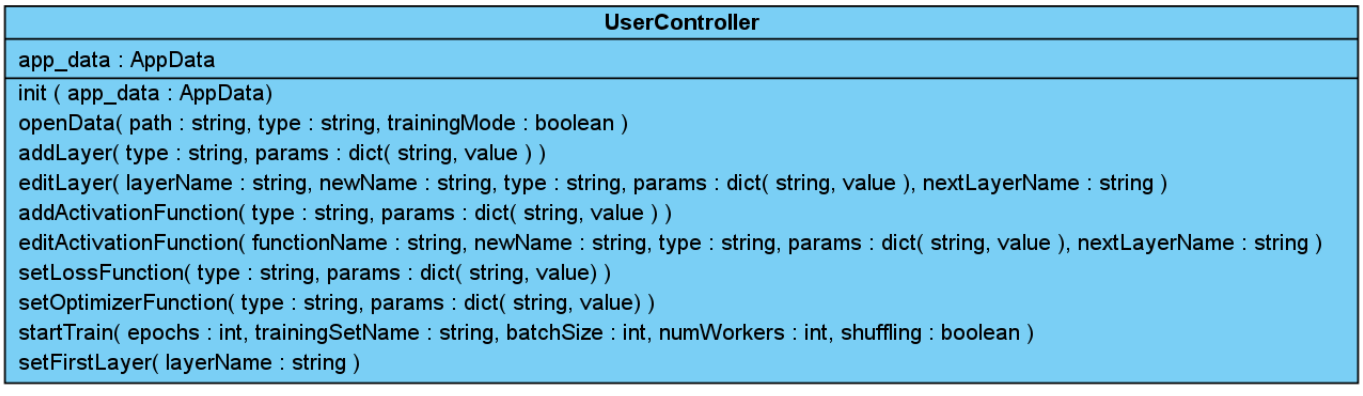


Figure 3.4: Controller class diagram

3.1 External Libraries

3.1.1 PyQT4

Our application was mainly developed with PyQT4 a Python module wich allow usage of QTDesigner (version 4). QT designer is a graphical tools that let developer build Graphical

Interfaces. At the opening, developer as the choice about the type of window he want to begin with:

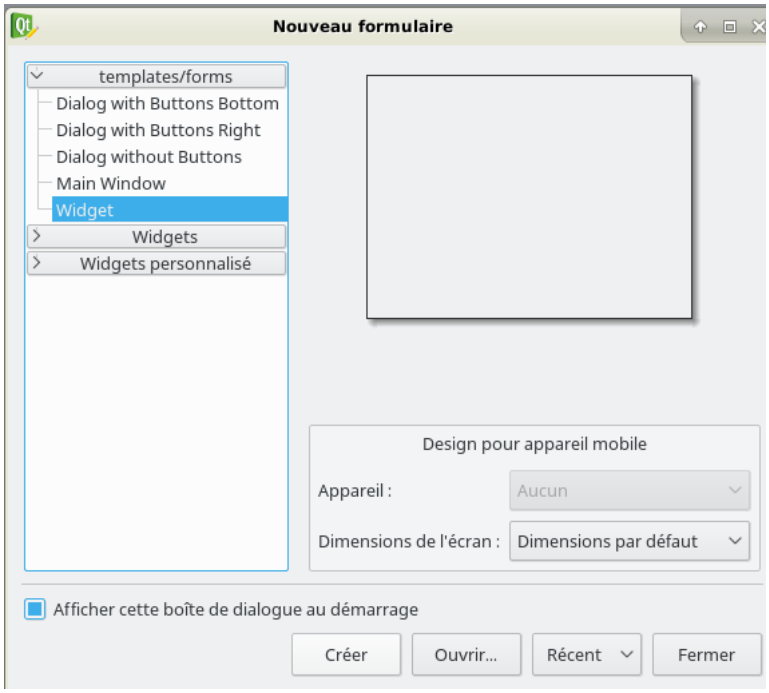


Figure 3.5: New form

In our case, we chose to start with a blank "Main Window". This setting generate us our QMainWindow Class wich contains all of our widgets, layouts, buttons, graphics view, menu and status bar.

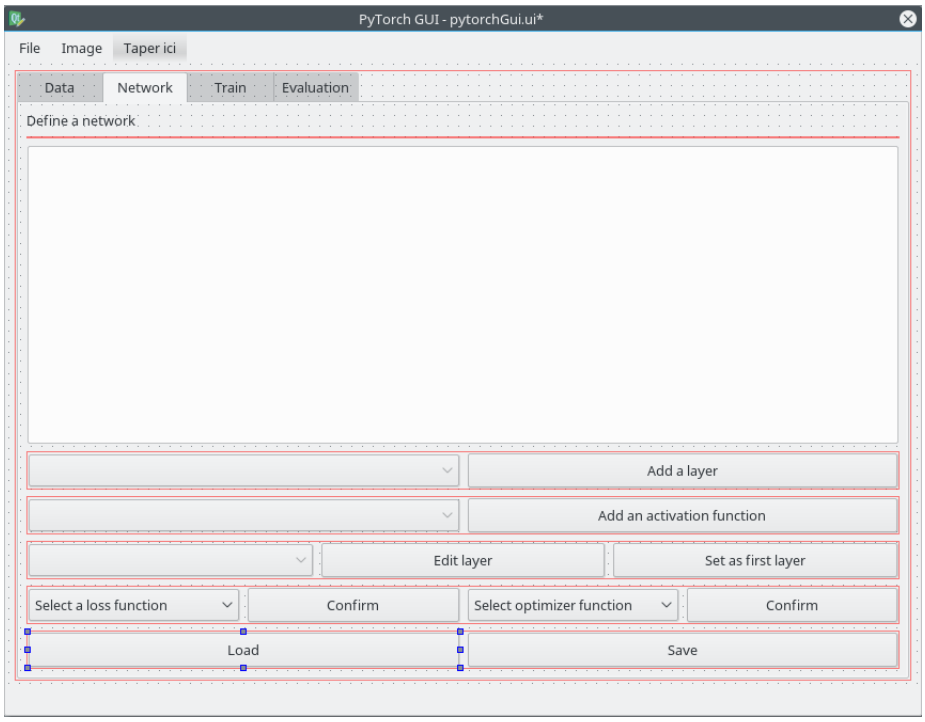


Figure 3.6: Component assembly in the Main Window

For each of this objects, their classes are automatically generated and can be available in the "Main Window" attributes.

Figure 3.7: Class representation of a "Load" button in QT Designer

Signals and sloats are used for communication between objects, it's a central feature of Qt. All of our buttons are connected with functions.

Other libraries

We will use some libraries that will ease us the task. Those libraries might change over the progress of our project :

- qtnodes: qtnodes is a library that helped us to do the Node graph visualization and editing with PySide.
- PySide: A Python binding of Qt.
- Plotly: Using this library could be very useful. It can indeed provide a lot of graphical features, like plotting data, and generating graphs.
- Numpy: Numpy is an extension to the Python language, which allows to manipulate matrices and multidimensional arrays, and to use a variety of mathematical operations on it. Numpy is already integrated to PyTorch.
- Matplotlib: Plotting library for Python used to build and display graphs.
- Pydot: Python interface to Graphviz's Dot.
- Networkx: A python library for studying graphs and networks.
- Tqdm : Library used for display progress bar on terminal while training or evaluation of the neural network is in progress.
- SetupTools: It's a package development process library designed to facilitate packaging Python projects by enhancing the Python standard library distutils.
- Pandas: Library used for data manipulation and analysis.
- Appdirs: A small Python module for determining appropriate platform-specific directories.

Utilities

- Version control system (VCS) As we work as a group and not individually on totally independent tasks, we have to use a VCS. Since we all know GitHub, we chose to use this simple VCS.
- UML designer :
  - VisualParadigm Online
  - ArgoUML

To clarify what we had to do to and move forward on the project, we needed to make a UML diagram. We first worked on VisualParadigm Online since we can all modify it then we finished it on ArgoUML.

3.1.2 Loading of the UI

A really interesting feature of PyQT to facilitate the development of graphical interface is the ability to load a **.ui** file in the application. This file contains all the graphical structure of our application. It can be edited manually, but most importantly it can be created and edited from the application **QT Designer**. Therefore, with a practical interface, we can place all the graphical elements of our application, and place them into layouts. Layouts allow us to organize the display elements in an organized and aligned way. We can parameter all the fields, as well are their names. This kind of loading makes the development of this type of application easier and more flexible, since we don't have to directly modify code in order to edit most of the view elements.
Once this file is loaded in our main window class, we have access to all the field with the corresponding variable name we chose when creating the UI file, and we can set values, as well as creating events, to program interaction with the user.

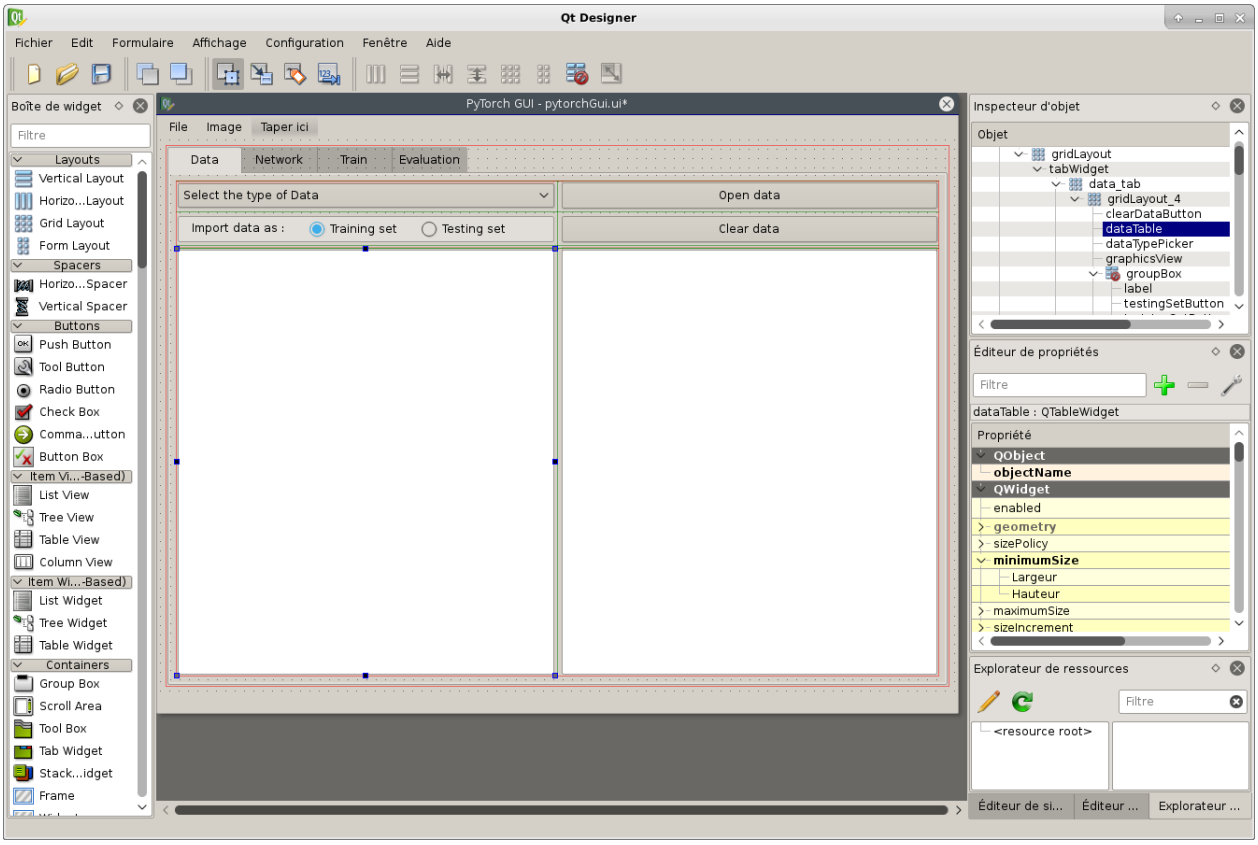


Figure 3.8: A screenshot of the QT Designer application

3.1.3 Organisation of the network as a graph

The main idea of a neural network is to be able to execute it by computing its layers in a certain order. This means layers follow each other until there is no **next layer**.

Implementation

To implement this graph in our code, we started by adding a python dictionary in our Network class. This dictionary is called **LayerList**, and it contains all the layers added in the network. We also have a second dictionary called **ActivationFunctionList** which contains all the added activation functions.

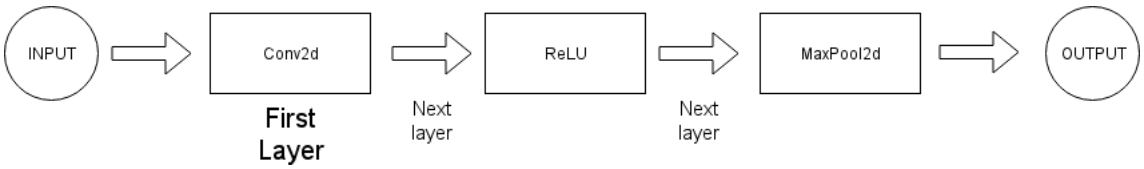


Figure 3.9: An example of a graph we can create with our current implementation

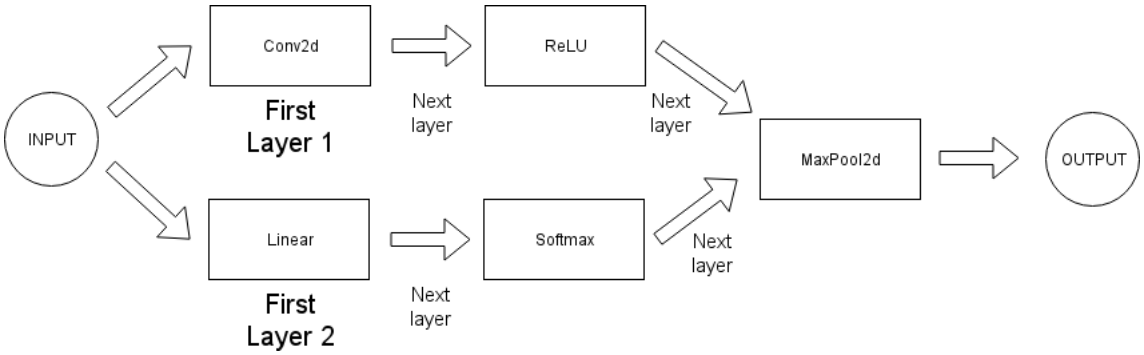


Figure 3.10: An example of a graph we could create by implementing parallel execution

Note : both of these dictionaries contain entries to the type **Layer**, as in fact an activation function **is a layer**. We just decided to separate them in two different dictionaries, to facilitate the implementation of our graphical interface.
Then, after adding the dictionaries to store our layer, we have to link layers between each other. To do so, we thought about a few possibilities. One of them was to create another dictionary, which would contain pairs of layers, which would mean they follow each other.
To facilitate the iteration through our graph, we instead decided to set an attribute called **nextLayer** in our layers. That way, we can directly get the layer that follows the layer we just computed.

Possible improvement : parallel execution

A possible improvement we discussed about for our graph was to implement a way to execute multiple chains of layers at the same time. This happens to be useful in specific cases. To implement this we would have to create, among other things, a list of **first layers**, which would all take a copy on the input data, to execute them in parallel, and to make them merge into other layers, in order to have **only one output** at the end.

3.1.4 Usage of JSON files to store data

Why we used this format

To be used as intended, our application has to store data on the disk.
To begin, we have to store default values which are necessary to load our application at the beginning. These values are :

- Available layer types in PyTorch, their parameters, the type of these parameters, as well as their default value
- Available activation functions (same thing)
- Available optimizer functions (same thing)
- Available loss functions (same thing)

We need all these informations, in order to display them to the user, and to allow them to choose a function or a layer, and parameter it in relation to the parameters accepted by PyTorch.
In addition to that, we also have to be able to store the network created by the user. When the user wants to save the network he created, we have to generate a file which will describe all the structure of our network. Our application also has to allow the user to load a previously saved network from a file. Since this structure is a graph, we need a file structure which will allow us to easily describe its nodes and their relations.

Consequently, to fulfill all these needs, we needed a file format which would allow us to store data with a system of nodes. To do this, we had two main options : the XML format, and the JSON format. We decided to use a **JSON file format** to store our data, and the reasons for this choice are :

- It is easier to parse than XML. A single line of code is needed to convert the content of the file to a python dictionary.
- It is lighter. The file takes less place, because there is no end-tag for each value. This is clearly valuable for us, since we have a great amount of default PyTorch functions to parse, it makes the file easier to read, and to modify.

How we organized our JSON files to store default functions

To store the default types of layers and functions available in PyTorch, we decided to organize our file as shown on the picture. We have a root which contains entries, and their names are the names of the associated functions in PyTorch. The values of these entries are other entries, which represent the different parameters of the function. The keys of these entries are the names of the parameters. The values are a **type** value, which allows our interface to know which type of field to display, and a **default** value, which contains the default value of this parameter, if there is one.
Note : We based all our knowledge for these information on the official PyTorch documentation.

How we load these default functions in our application

To load these default functions, we have to parse the JSON file, and load the data in Python. To do so, we created a class named **FunctionDatabase**. This class is responsible for loading all the default functions, by loading the content of the JSON file, and putting it into a python Ordered Dictionary. This dictionary will contain other dictionaries, which describe all the available parameters for the corresponding function. Default python dictionaries are not ordered, which means that by loading our default parameters to display them in the application, they might not appear in the right order. That's the reason why we used an OrderedDict, instead of a regular python dictionary. That way, all our available layers and functions, as well as their parameters will always be displayed in the same order, that means the order in which they have been written in the file.
We use **a total of four FunctionDatabase**. One for the layers, one for the activation functions, one for the loss functions, and one for the optimizer functions. They are all stored in the **AppData** class, so they can be called and displayed from the view. The corresponding JSON files are loaded once at the start of the application, then we never use them again.

How we organized our JSON files to save and load created networks

In order to store a created network, we also created our own format. A way to save and load a network already exists in PyTorch. But it doesn't fit our needs. Firstly, it's encrypted in

Figure 3.11: A parsing of our JSON file for storing available layers in PyTorch

Figure 3.12: A parsing of our JSON file for storing a saved network

a way that it is only readable by PyTorch, so that a user can't read it and modify it manually. Also, it doesn't allow us to save the order of execution of our layers, which is a key part of how we designed our networks. These are the reasons why we decided to, once again, create our own format.
As you can see of the picture, our saved file is organized like this :

- The root contains keys, which are the names of the different layers (or activation functions).
- Each of them contains information about the layer : its **class** ("layer" or "activation function"), its **type**, and its **parameters**.
- We also added a field containing the name of the **next layer**. This allows us to save our network **as a graph**.
- Lastly, we add an optional field named **firstlayer**, which tells us if the layer is the first layer of the network. It means that when training the network, it will start by computing this layer, and then the others.

How we save a network to a JSON file

In order to save our network to the JSON format, we first need to create the dictionary that will represent it. As always, we create an Ordered Dictionary, thus we can save everything in a predictable order.
In order to store these information in the dictionary, we start by iterating through all the added layers, and we save them with the class "layer", their type, their parameters, the name of their

next layer (if they have one), and a "firstLayer" value if it is the first layer of the graph. Then, we do the same by iterating through the added activation functions. The only difference here is that an activation function cannot be the first layer, so we put this possibility aside.
When we finished to put the data we want to store in the dictionary, we just dump it directly to the JSON file.

How we load a network from a JSON file

The loading of a network from a json file works as follows :

1. We clear the list of layers, and the list of activation functions, because we want to load a new network, without the layers from the old one
2. We parse the json file, by converting it into an Ordered Dictionary
3. We iterate through every node, and we create a Layer from the information stored in it (name, type, parameters).
4. If the class is "layer", we add it to the layer list, and we check if it is the first layer of the network. If the class is "activation_function", we add it to the activation function list.
5. Once all our layers and functions are added to the lists, we iterate again through every node, to check which layer goes after the other, and we update these information on what we added before, to reconstitute our graph.

Loading of layer types and functions types in the view

By using the different **FunctionDatabase** we loaded at the starting of the app, the view begins by loading the different layer types, activation function types, loss function types and optimizer function types in combo boxes. That way, the user can select the desired type, and initiate actions related to the selected type by clicking on the related buttons.

Generating forms to add and edit network layers and functions

When the user wants to parameter the network, he will want to set the different parameters of functions and layers. To do so, we generate dynamical forms, which parameters vary according to the type selected by the user. We take into account the type of parameter to generate a relevant field. For example, the type "number" will generate a number selector, and the type "boolean" will generate a checkbox.
We generate this kind of forms for the following actions :

- Adding a layer
- Adding an activation function
- Editing a layer
- Editing an activation function
- Setting the loss function
- Setting the optimizer function

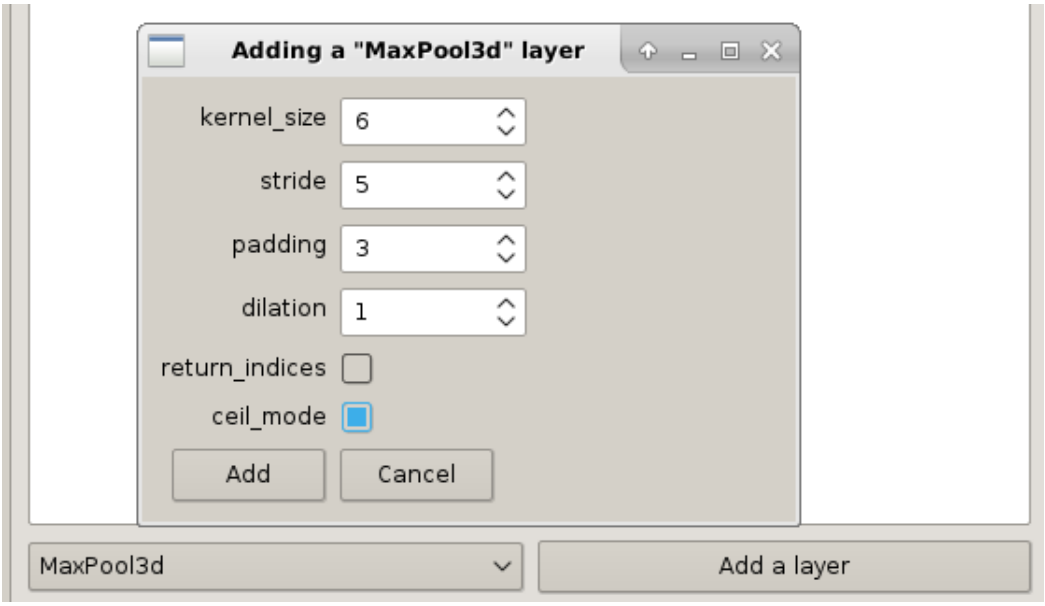


Figure 3.13: An example of a generated form to add a layer to the network

If the user closes the form, nothing happens. If he validates it, the information is transmitted to the model, which creates (or modifies) the layer (or the function) with the corresponding parameters.
When editing a layer or an activation function, two fields are added to the original parameters. These two fields are a text input for the name of the layer/function, and a combo box, which allows the user to select the next layer. It means that after the currently edited layer has been computed, it will then compute the layer selected in this field. By default, its value is **None**, so in this case the computation will stop here.

Chapter 4

Functional analysis and tests

4.0.1 Application Features

Graphic User Interface with PyTorch

The purpose of this project(PyTorch GUI) is to develop a Python-based GUI(Graphical User Interface) which uses PyTorch Libraries and its an open-source deep learning graphic user interface such as Expresso for Caffe, NVidia DIGITS for Theano, Caffe, Torch and Tensorflow.

Even though there are many graphical user interfaces which provide numerous tools for deep learning purposes but unfortunately they are not compatible with PyTorch Library at the moment we began this project. Therefore, we proposed an application that is an graphical user interface for PyTorch and provides the abilities to use all deep learning features provided by the Pytorch framework.

PyTorch GUI provides a complete featured deep learning application with variety of options. The **Qt4 Designer** is used for front hand in this project which provides all graphical objects such as Form, Tabs, Buttons, Drop-down-menus, Radio-buttons, Check-boxes etc. For the back hand, Python language with PyTorch framework are used to import, create the network, train, and classify the datasets. All features of this application is

The first Tab is **"Data"** Tab, which is responsible for importing datasets for further operations.

In this Tab we have the following options:

1. Drop-down list
   Contains different datasets such as CIFAR10, CIFAR100, MNIST. These datasets holds different types of images for training in various image processing systems and Machine learning. The user has to select the type data he will import.

2. Training set
   User may check this option in order to train and classify the data imported.

3. Testing set
   User may check this option in order to Test whether our Train data is closest to the accurate result or not.

4. Open data
   The user can import data via this button. It will be then displayed in the "Data table" on the left.

5. Clear data
   Will clear imported data in case if the user want to import another type of data.

6. Data table
   The imported dataset will be displayed in the following "Data Table".

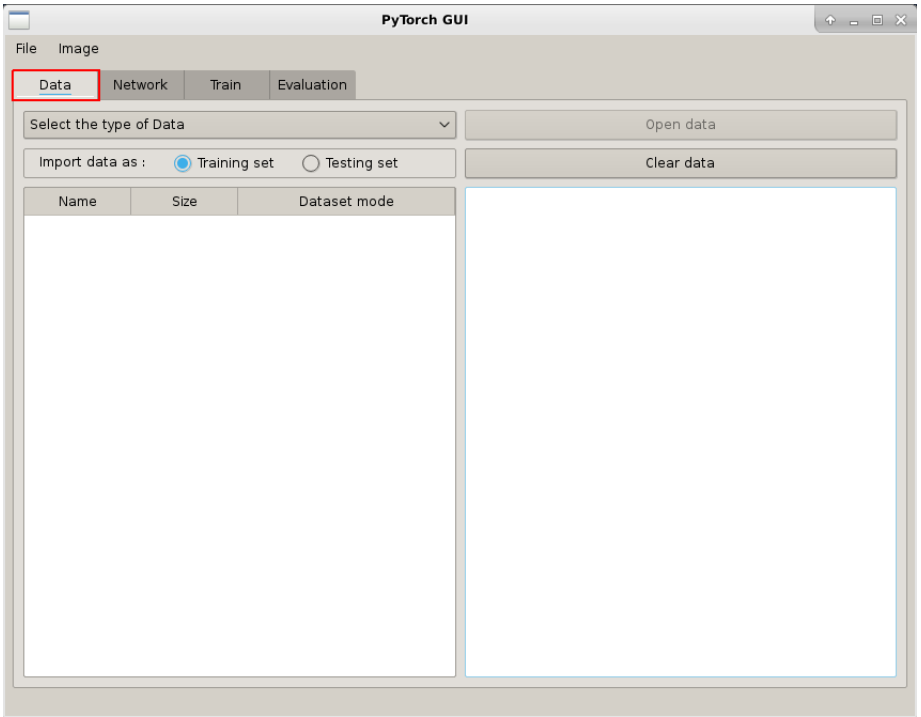


Figure 4.1: Data Tab

The Network tab enables the users to choose their desire neuron network according their needs.

There are the following options and objects in this tab:

1. This widget is used to show the graph of the network.

2. Select a layer
   The first drop-down (from left side) allows the users to select the layer such as Conv1d, Conv2d, Conv3d and so on.

3. Add a layer
   Once a layer is selected it will display a pop-up where the user can edit the layer's parameters. He will then have to clicked on **add** to add the layer into the current network.

4. Select an activation layer
   The next down (drop-down list) is used to select activation function for example ReLU, ReLU6, ElU, Tanh and so on.

5. Add an activation function
   Once an activation function is selected it will display a pop-up where the user can edit the parameters, he will then click on **Add** to add the function to the current network.

6. Select a layer
   In this drop-down, the user can select an existing layer of the network.

7. Edit layer
   Once a layer is selected in the previous drop-down, the user can edit the layer (rename, change parameters, set a nextLayer)

8. Set as first layer
   Once a layer is selected in the previous drop-down the user can set it as the first layer of the neural network.

9. Select a loss function
   In this drop-down, we choose a loss function.

10. Confirm
    Once a loss function is selected, it will open a pop-up where user can edit the parameters. He will then have to click on **set** to validate the function.

11. Select optimizer function
    In this drop-down the user can select an **Optimizer function** (load from a database in background).

12. Confirm
    Once an optimizer function is selected, it will display a pop-up where the user can edit the parameters. He will then have to click on **Set** to validate the function.

13. Load
    It opens a file-explorer in which the user can only select json files, if the file is correct it will load the layers that composed a neural network and put them into our current network dictionary layers.

14. Save
    The users can save the current dictionary that contains all the layers that compose the current neural network into a json file, a file explorer will display and the user can choose the Path he want to save his file.

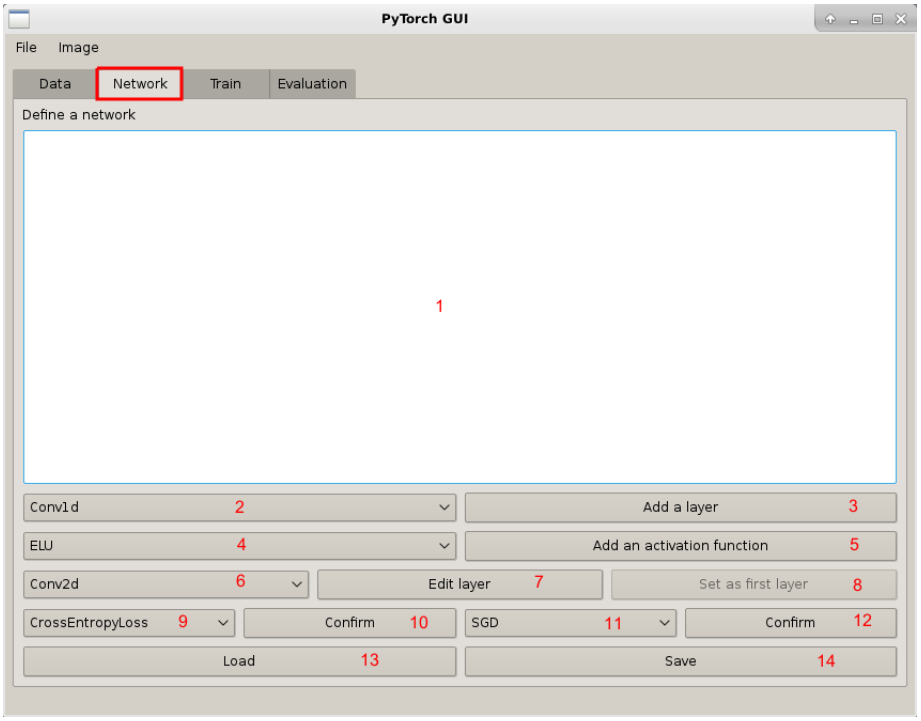


Figure 4.2: Network Tab

The Train tab is used to Train and classify the imported datasets There are the following options and objects in this tab:

1. Train set
   In Training set we select the type of dataset that we have already chose such as CIFAR10 or any other type.

2. Number of epochs
   Number of epochs are the number of iterations that we train the dataset.

3. Batch size
   The whole dataset will be divided according to the given number to Batch size and each divided group would be processed by a thread.

4. Worker number
   Divides the process on all of the core exists on the machine where this application might run.

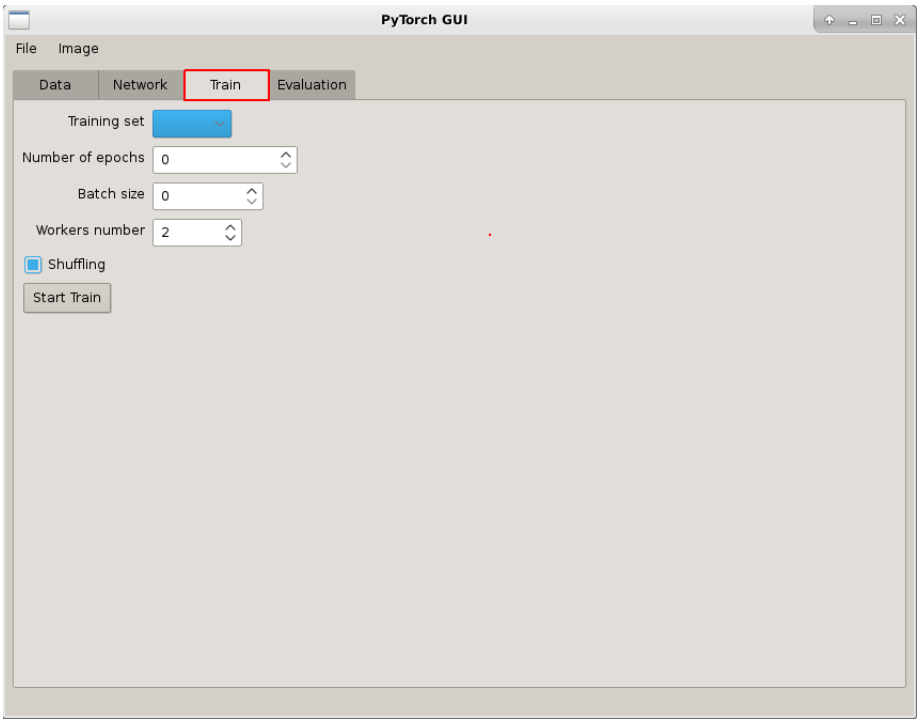


Figure 4.3: Train Tab

In this tab the end result will be evaluated to check that how far our result is from desired value and we have the following objects: This tab for the moment is not working.

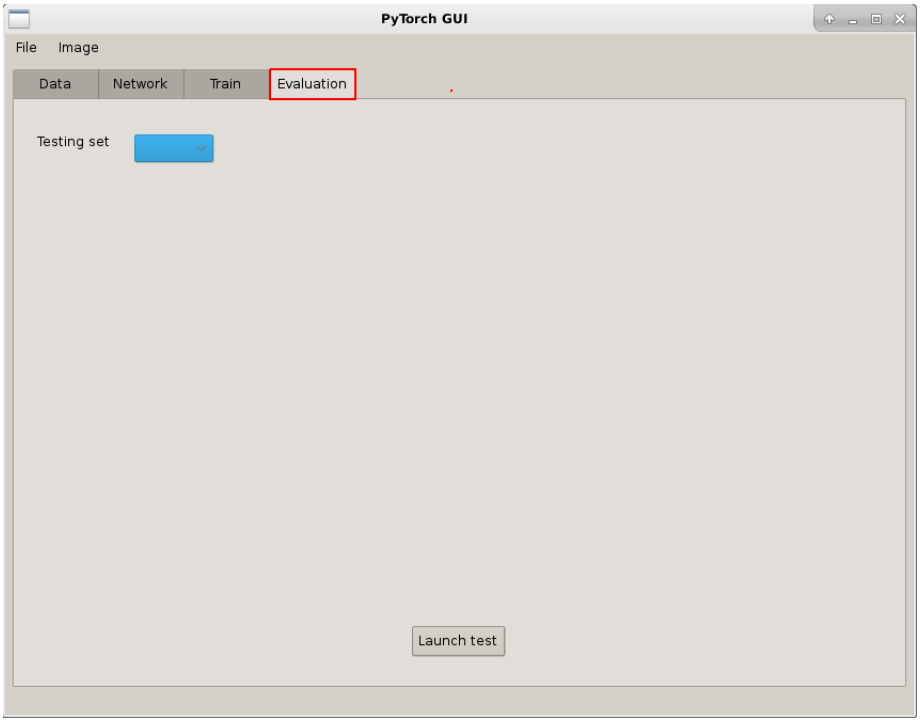


Figure 4.4: Evaluation Tab

4.0.2 Edit neural network from scratch

Users have two different way for building a Neural Network. Etheir from scratch or from an pre-loaded model.

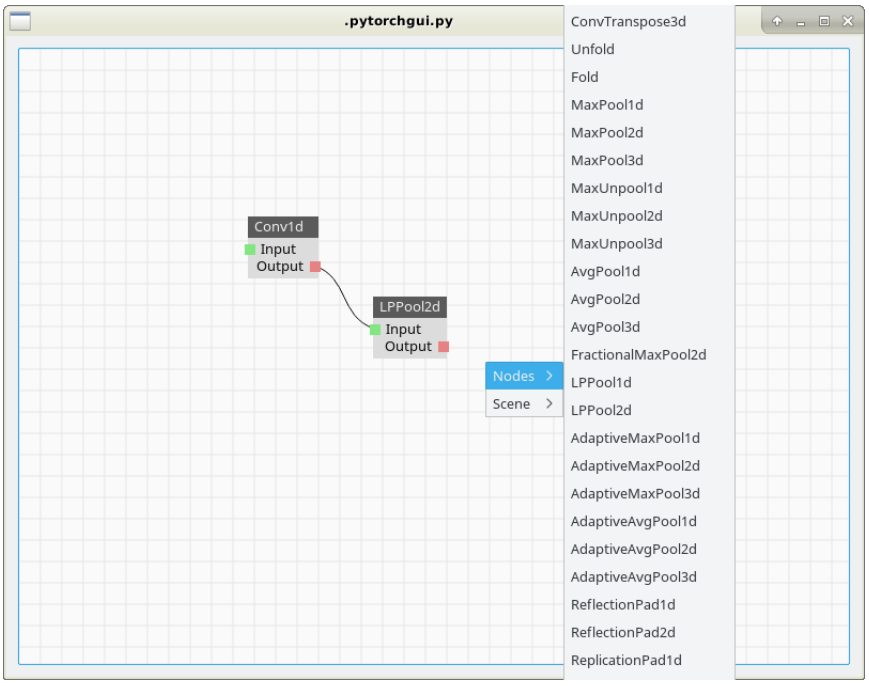


Figure 4.5: From scratch feature

From scratch user just have to click on add a layer and he can see the graph representation.

4.1 Problems of the application

4.1.1 Non-working features

Some features are not totally implemented:

- Load json from scratch
- Evaluate
- Forward function in training, we have to code it

4.1.2 Non-implemented features

Some features are almost not implemented, so their functionalities are not currently working

- Fetch the layer type set in the json files and pass them to the scratch module
- Load a model from the scratch module and show the graph representation

The Evaluation Class

The evaluation methods are already present in the Evaluation class, but we didn't include them in the rest of the application

Training result display

The display of the training is currently printed in the console. An improvement of this could be to print graphs in the application.

Exception handling

There are two types of exceptions that are returned by the application :

- our own exceptions, from our application
- exceptions from PyTorch

For the moment, all the exceptions are printed in the console. To handle them, we could notify the user by displaying a popup in the graphical interface.

4.2 Test

To make sure the application is working properly, we need to test all features with determined data sets.

Examples of tests :

- Importing a set of data, edit its variables and export it.
  To test this functionality we can simply try do import a data supported by Pytorch. We can use the Pytorch tutorials to test the importation of data like Cifar10. We import as a training set the Cifar10 dataset, it will create a data object of size 50000, with a data mode of **Testing**.
  We can see if the data are correct by checking to their parameters, if they are of the form:

Data Name	Data	batch_size	shuffle	num_workers
trainloader	trainset	batch_size=4	shuffle=True	num_workers=2
testloader	testset	batch_size=4	shuffle=False	num_workers=2

- Load a network from a json file.
  We can check if all the layer in the json file are correctly add to the network by checking if all the layer's name and parameters are also in the network. To do this we can directly compare if all the keys and object in the network's **layerList dictionary** is equal to the dictionary created by the json file. Because we need the parameters in a certain order we use **OrderedDictionary**, if the save json file is not correctly write we can have error during the build of the network.

- Save a network in a json file.
  To save the network we use the json function **write**, to save the network **layerList** dictionary. We can check if the file is correctly saved by trying to load it and build the network.If we got an error that means one parameter is not correctly write or is not at the good place.

- Editing a layer
  To check if the layer is correctly change, we can check by testing if the layer's parameters are equal to the new one we set.

- Renaming a layer
  To check if the layer have been correctly renamed, we can check if a layer with the new name defined by the user is present in the network layerList dictionary. We also need to check if a layer with the old name is not present into it too.

- Remove a layer
  To test if a layer have been correctly remove, we can check if her name is still present into the network **layerList** dictionary.

- Testing that the results of classification are always the same as when we use PyTorch in console mode. It's impossible to get exactly the same result in our application and by using Pytorch in console mode because of the random variables of the DeepLearning. But we can compare if the results are close or not. So the test To do this we can compare the running_loss that we got during the training by comparing the graphic that we obtain in the **resultTraining** directory. We can also compare the result of the result of the evaluation, for an image classifier we obtain percentages of the the good prediction of the network on the tested data.

- Ergonomics and stability : we have to test that all buttons do an action, that they do the right action, that the application does not crash, and that it is well integrated to the operating system

Chapter 5

Conclusion

5.0.1 The project

This project was interesting for many reasons. It made us work on many part of the development of a modern application. In fact, we had to work on the designing of graphical interface, and linking it to a model, which calls an already existing and complex application. This made us learn a lot of things about making graphical interfaces, but also things on the very complex and modern subject of deep learning.

5.0.2 Difficult parts

The complexity of this project was also what made it very challenging for us. In fact, it took us a lot of time to understand clearly what were the needs and requirements, and most importantly to understand the subject of deep learning itself. This made us take a long time to organise our architecture, and consequently to start to code.

Bibliography

[1] Les bases de pyqt. https://tcuvelier.developpez.com/tutoriels/pyqt/bases/.

[2] Les bases de pyqt. https://tcuvelier.developpez.com/tutoriels/pyqt/bases/.

[3] Mikael Laine. Back propagation image from a video. https://www.youtube.com/watch?v=8d6jf7s6$_Q$s.

[4] Assaad MOAWAD. Neural networks and back-propagation explained in a simple way. https://medium.com/datathings/neural-networks-and-backpropagation-explained-in-a-simple-way-f540a3611f5e.

[5] Anish Singh Walia. Neural networks and back-propagation explained in a simple way. https://towardsdatascience.com/types-of-optimization-algorithms-used-in-neural-networks-and-ways-to-optimize-gradient-95ae5d39529f.