Convolutional Neural Networks

GSoC project: GNU Octave community

Author:
Enrico Bertino

26th February 2018
# Contents

1 Introduction 2

2 Theoretical dive 2
   2.1 Fundamentals ........................................... 2
   2.2 Convolutional neural networks ......................... 6
   2.3 CNN layers ............................................. 6
   2.4 Training options ..................................... 9

3 Implementation 10
   3.1 Tensorflow overview .................................... 10
   3.2 Octave .................................................. 11
      3.2.1 Layers ............................................ 11
      3.2.2 TrainNetwork ..................................... 12
      3.2.3 SeriesNetwork ................................... 13
   3.3 Python .................................................. 13

4 The code 13
   4.1 Code repository ......................................... 13
   4.2 Commits .................................................. 13
   4.3 Testing Tensorflow installation ....................... 14
   4.4 Layers in Octave ....................................... 15
      4.4.1 Layer class ...................................... 15
      4.4.2 Layers class ..................................... 16
   4.5 Layers in Python ....................................... 18
   4.6 TrainingNetwork in Python ............................. 19
   4.7 Brief tutorial .......................................... 20
   4.8 Example: CNN for regression ......................... 20
   4.9 Troubleshooting ........................................ 22

5 Future improvements 24
1 Introduction

I present here the project developed in collaboration with the GNU Octave community about Convolutional Neural Networks. GNU Octave is a high-level language, primarily intended for numerical computations, which provides a convenient command line interface mostly compatible with Matlab. On the other side, the machine learning community is growing up and there is a growing demand for the implementation of deep learning methods. Convolutional neural networks are nowadays widespread in the world of image recognition thanks to their ability to extract meaningful features simply starting from data. In recent years, there is a strong tendency of open-researching in the deep learning field, where even the biggest companies open their code and many open-source frameworks have been created. These machine learning communities focus their attention on Python, since it is becoming the most used language in the data science world. This is mostly due to its ease of syntax and the fact that, although it is not a compiled language, it maintains a good computational speed since it is based on numerical libraries written in C++, such as Numpy. This led me to choose to write the "backend part" of the package in Python, using the Tensorflow API. This integration allows me to guarantee a continuous updating on neural networks improvement also in the future and leading to a greater focus on Octave interface. In this sense, letting Octave coexist with Tensorflow, I exploited their synergy reaching fast consistent goals. The project was sponsored by the Google Summer of Code, a global program that every year brings student developers into open source software development.

2 Theoretical dive

The theoretical part of this project was based on all the concepts of deep learning. In particular [1], [7], [8] and [9] were very useful in the whole path. I present in this Section the main notions underlying the neural networks.

2.1 Fundamentals

I want to start with a brief explanation about the perceptron and the back propagation, two key concepts in the artificial neural networks world.

Neurons. Let’s start from the perceptron, that is the starting point for understanding neural networks and its components. A perceptron is simply a "node" that takes several binary inputs, \( x_1, x_2, \ldots \), and produces a single binary output:

\[
\text{The neuron’s output, } 0 \text{ or } 1, \text{ is determined by whether the linear combination of the inputs } \omega \cdot x = \sum_j \omega_j x_j \text{ is less than or greater than some threshold value. That is a simple mathematical model but is very versatile and powerful because we can combine many perceptrons and varying the weights and the threshold we can get different models. Moving the threshold to the other side of the inequality and replacing it by what’s known as the perceptron’s bias, } b = -\text{threshold, we can rewrite it as}
\]
Using the perceptrons like artificial neurons of a network, it turns out that we can devise learning algorithms which can automatically tune the weights and biases. This tuning happens in response to external stimuli, without direct intervention by a programmer and this enables us to have an "automatic" learning.

Speaking about learning algorithms, the proceedings are simple: we suppose we make a small change in some weight or bias and what see the corresponding change in the output from the network. If a small change in a weight or bias causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want. The problem is that this isn’t what happens when our network contains perceptrons since a small change of any single perceptron can sometimes cause the output of that perceptron to completely flip, say from 0 to 1. We can overcome this problem by introducing an activation function. Instead of the binary output we use a function depending on weights and bias. The most common is the sigmoid function:

$$\sigma(\omega \cdot x + b) = \frac{1}{1 + e^{-(\omega \cdot x + b)}}$$

With the smoothness of the activation function $\sigma$ we are able to analytically measure the output changes since $\Delta out$ is a linear function of the changes $\Delta \omega$ and $\Delta b$:

$$\Delta out \approx \sum_j \frac{\partial out}{\partial \omega_j} \Delta \omega_j + \frac{\partial out}{\partial b} \Delta b$$

**Loss function** Let $x$ be a training input and $y(x)$ the desired output. What we’d like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all $x$. Most used loss function is mean squared error (MSE):
\[ L(\omega, b) = \frac{1}{2n} \sum_x ||Y(x) - \text{out}||^2, \]

where \( n \) is the total number of training inputs, \( \text{out} \) is the vector of outputs from the network when \( x \) is input.

To minimize the loss function, there are many optimizing algorithms. The one we will use is the gradient descend, of which every iteration of an epoch is defined as:

\[
\omega_k \rightarrow \omega_k' = \omega_k - \eta \sum_j \frac{\partial L}{\partial \omega_k},
\]

\[
b_k \rightarrow b_k' = b_k - \eta \sum_j \frac{\partial L}{\partial b_k},
\]

where \( m \) is the size of the batch of inputs with which we feed the network and \( \eta \) is the learning rate.

**Backpropagation** The last concept that I would like to emphasize is the backpropagation. Its goal is to compute the partial derivatives \( \frac{\partial L}{\partial \omega} \) and \( \frac{\partial L}{\partial b} \) of the loss function \( L \) with respect to any weight or bias in the network. The reason is that those partial derivatives are computationally heavy and the network training would be excessively slow.

Let be \( z^l \) the weighted input to the neurons in layer \( l \), that can be viewed as a linear function of the activations of the previous layer: \( z^l = \omega^l a^{l-1} + b^l \).

In the fundamental steps of backpropagation we compute:

1. the final error:
   \[
   \delta^L = \Delta a_L \odot \sigma'(z^L)
   \]
   The first term measures how fast the loss is changing as a function of every output activation and the second term measures how fast the activation function is changing at \( z^L \).

2. the error of every layer \( l \):
   \[
   \delta^l = ( (\omega^{l+1})^T \delta^{l+1} ) \odot \sigma'(z^l)
   \]

3. the partial derivative of the loss function with respect to any bias in the net
   \[
   \frac{\partial L}{\partial b^l_j} = \delta^l_j
   \]

4. the partial derivative of the loss function with respect to any weight in the net
   \[
   \frac{\partial L}{\partial \omega^l_{jk}} = a^{l-1}_k \delta^l_j
   \]
We can therefore update the weights and the biases with the gradient descent and train the network. Since inputs can be too numerous, we can use only a random sample of the inputs. Stochastic Gradient Descent (SGD) simply does away with the expectation in the update and computes the gradient of the parameters using only a single or a few training examples. In particular, we will use the SGD with momentum, that is a method that helps accelerate SGD in the relevant direction and damping oscillations. It does this by adding a fraction $\gamma$ of the update vector of the past time step to the current update vector.

**Neural networks** An Artificial Neural Network (ANN) is a computational model that is inspired by the way biological neural networks in the human brain process information. Artificial Neural Networks have generated a lot of excitement in Machine Learning research and industry, thanks to many breakthrough results in speech recognition, computer vision and text processing. The definition of a neural network, is provided by the inventor of one of the first neurocomputers, Robert Hecht-Nielsen, who defines a neural network as a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs. Roughly speaking, neural network is a set of connected neurons whose connections are regulated by weights (like synapses in biological neurons). When we train a neural network we want the neurons to fire whenever they learn specific patterns from the data, and we model the fire rate using an activation function.

Let’s define the basic elements of a NN:

- A block of nodes (the perceptrons) is called **layer**
- **Input** Nodes (input layer): no computation is done here within this layer, they just enter the information in the network.
- **Hidden** nodes (hidden layer): in Hidden layers is where intermediate processing or computation is done, they perform computations and then transfer the weights (signals or information) from the input layer to the following layer. It is possible to have a neural network without a hidden layer.
- **Output** Nodes (output layer): here we use an activation function that maps to the desired output format (e.g. softmax for classification).
- Connections and weights: the network consists of connections, each connection transferring the output of a neuron $i$ to the input of a neuron $j$. In this sense $i$ is the predecessor of $j$ and $j$ is the successor of $i$, each connection is assigned a weight $W_{ij}$.
- **Activation** function: the activation function of a node defines the output of that node given an input or set of inputs. It is the nonlinear activation function that allows such networks to compute nontrivial problems using only a small number of nodes. This function is also called the transfer function. Most used activation functions are Sigmoid, Tanh, ReLU, leaky ReLU.
Learning rule: The learning rule is a method which modifies the parameters of the neural network, in order for a given input to the network to produce a favored output (e.g. the Stochastic Gradient Descent).

2.2 Convolutional neural networks

Convolutional Neural Networks (CNNs) are very similar to ordinary Neural Networks, they are made up of neurons that have learnable weights and biases. In convolutional neural network the unit connectivity pattern is inspired by the organization of the visual cortex: units respond to stimuli in a restricted region of space known as the receptive field. Receptive fields partially overlap, over-covering the entire visual field. Unit response can be approximated mathematically by a convolution operation. They are variations of multilayer perceptrons that use minimal preprocessing. Their wide applications is in image and video recognition, recommender systems and natural language processing.

For image recognition, usually the input (the image) is a 3D-array $\text{Width} \times \text{Height} \times \text{ColorChannels}$ (usually color channels are 3 for an rgb image). Common layers used in a CNN are: Convolution, ReLu, Pooling as Fully connected (Fig. 2).

2.3 CNN layers

Here a brief explanation of the layers used in CNNs. The names of the corresponding classes that I am considering are taken form the Matlab package [18]. All these classes inherits from the Layers class, as explained in Section 3.2.

Convolution2DLayer The convolution layer is the core building block of a CNN and it does most of the computational heavy lifting. They derive their name from the “convolution” operator. The primary purpose of convolution is to extract features from the input image preserving the spatial relationship between pixels by learning image features using small squares of input data.

In the example in Fig. 3, the 3x3 matrix is called a 'filter’ or 'kernel’ and the matrix formed by sliding the filter over the image and computing the dot product is called the 'Convolved Feature’ or 'Activation Map’ (or the 'Feature Map’). In
practice, a CNN learns the values of these filters on its own during the training process (although we still need to specify parameters such as number of filters, filter size, padding and stride). The more number of filters we have, the more image features get extracted and the better our network becomes at recognizing patterns in unseen images.

The size of the Feature Map depends on three parameters: the depth (that corresponds to the number of filters we use for the convolution operation), the stride (that is the number of pixels by which we slide our filter matrix over the input matrix) and the padding (that consists in padding the input matrix with zeros around the border).

**ReLU Layer**  ReLU stands for Rectified Linear Unit and is a non-linear operation: $f(x) = \max(0, x)$. Usually this is applied element-wise to the output of some other function, such as a matrix-vector product. It replaces all negative pixel values in the feature map by zero with the purpose of introducing non-linearity in our network, since most of the real-world data we would want to learn would be non-linear.

**Fully Connected Layer**  Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Hence their activations can be computed with a matrix multiplication followed by a bias offset. In our case, the purpose of the fully-connected layer is to use these features for classifying the input image into various classes based on the training dataset. Apart from classification, adding a fully-connected layer is also a cheap way of learning non-linear combinations of the features.
Pooling  Neurons in a fully connected layer have full connections to all activations in $t$. It is common to periodically insert a pooling layer in-between successive convolution layers. Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map but retains the most important information. In particular, pooling

- makes the input representations (feature dimension) smaller and more manageable
- reduces the number of parameters and computations in the network, therefore, controlling overfitting
- makes the network invariant to small transformations, distortions and translations in the input image
- helps us arrive at an almost scale invariant representation of our image

MaxPooling2DLayer  In case of Max Pooling, we define a spatial neighborhood (for example, a $2 \times 2$ window) and take the largest element from the rectified feature map within that window.

AveragePooling2DLayer  Instead of taking the largest element we could also take the average.

DropoutLayer  Dropout in deep learning works as follows: one or more neural network nodes is switched off once in a while so that it will not interact with the network. With dropout, the learned weights of the nodes become somewhat more insensitive to the weights of the other nodes and learn to decide somewhat more by their own. In general, dropout helps the network to generalize better and increase accuracy since the influence of a single node is decreased.

SoftmaxLayer  The purpose of the softmax classification layer is simply to transform all the net activations in your final output layer to a series of values that can be interpreted as probabilities. To do this, the softmax function is applied onto the net inputs.

$$\phi_{\text{softmax}}(z^i) = \frac{e^{z^i}}{\sum_{j=0}^{k} e^{z^j}}$$
CrossChannelNormalizationLayer  Local Response Normalization (LRN) layer implements the lateral inhibition that in neurobiology refers to the capacity of an excited neuron to subdue its neighbors. This layer is useful when we are dealing with ReLU neurons because they have unbounded activations and we need LRN to normalize that. We want to detect high frequency features with a large response. If we normalize around the local neighborhood of the excited neuron, it becomes even more sensitive as compared to its neighbors. At the same time, it will dampen the responses that are uniformly large in any given local neighborhood. If all the values are large, then normalizing those values will diminish all of them. So basically we want to encourage some kind of inhibition and boost the neurons with relatively larger activations.

2.4 Training options

In addition to the network architecture, the result of the model will be influenced by the parameters set for the training. The names are taken from the trainingOptions object of [18] that contains all these parameters for the training. A brief explanation:

SolverName  Optimizer chosen for minimize the loss function. To guarantee the Matlab compatibility, only the Stochastic Gradient Descent with Momentum (‘sgdm’) is allowed

Momentum  Parameter for the sgdm: it corresponds to the contribution of the previous step to the current iteration. See [6] for more information

InitialLearnRate  Initial learning rate $\eta$ for the optimizer

LearnRateScheduleSettings  These are the settings for regulating the learning rate. If it the learning rate drops piecewise, two values have to be defined: rate factor (DropRateFactor) and drop period (DropPeriod)

L2Regularization  Regularizers allow to apply penalties on layer parameters or layer activity during optimization. This is the factor of the L2 regularization.

MaxEpochs  Number of epochs for training

Verbose  Display the information of the training every VerboseFrequency iterations

Shuffle  Random shuffle of the data before training if set to ‘once’

CheckpointPath  Path for saving the checkpoints

ExecutionEnvironment  Chose of the hardware for the training: ‘cpu’, ‘gpu’, ‘multi-gpu’ or ‘parallel’. The load is divided between workers of GPUs or CPUs according to the relative division set by WorkerLoad
OutputFcn Custom output functions to call during training after each iteration passing a struct containing:
- Current epoch number
- Current iteration number
- TimeSinceStart
- TrainingLoss
- BaseLearnRate
- TrainingAccuracy (or TrainingRMSE for regression)
- State

3 Implementation

Since the goal was making the entire package of neural networks, it would have been impossible (and even useless) to implement everything from scratch. I chose to leverage Tensorflow (TF), an open source software library for numerical computation using data flow graphs. The Python API is at present the most complete and stable. To take advantage of the python API, I used the Pytave package [14], a native Python calling interface for GNU Octave which was made stable during summer 2016.

Figure 4: In order to call Tensorflow Python API, we use the octave package Pytave with which we use Python functions

3.1 Tensorflow overview

TF is an open source software library for numerical computation using data flow graphs. Nodes in the graph represent mathematical operations, while the graph edges represent the multidimensional data arrays (tensors) communicated between them. The flexible architecture allows to easily deploy computation to one or more CPUs or GPUs.

Graph and session TF uses the concept of data flow graph (or diagram) as its foundation. TF separates the definition of computations from their execution. Generally, there are two steps to perform a computation in TF: build a graph first, then you use a session to execute the graph (precisely, the operations in the graph). A computational graph is a series of TF operations arranged into a graph of nodes. Each node takes zero or more tensors as inputs and produces a tensor as an output. One type of node is a constant, which takes no inputs and it outputs a value it stores internally. To execute any operation, the computational graph needs a session where the tensors and operations would be evaluated. During the training, TF uses variables, which are in memory buffers containing tensors, to hold and update parameters. They must be explicitly initialized and can be saved to disk during and after training. The easiest way to initialize all the variables of the graph is to add at the beginning of the session an operation that runs all the variable initializers. Another type of object is the placeholders, that is a way to define variables without actually defining the values to be passed to it when we create a computational graph.
3.2 Octave

In the Octave implementation, I had to write the classes for the layers and the functions for calling the Neural network execution in Python. A general schema in Fig. 3.2. The core of the package is composed by three parts:

- **Layers**: there are 11 types of layers that I defined as Octave classes, using classdef. These layers can be concatenated in order to create a Layer object defining the architecture of the network. This will be the input for the training function.

- **Training**: the core of the project is the training function, which takes as input the data, the layers and some options and returns the network as output.

- **Network**: the network object has three methods (activations, classify and predict) that let the user compute the final classification and prediction.

3.2.1 Layers

Every layer type inherits some attributes and methods from the parent class Layers (Fig. 6). This is useful for creating the Layer object: the concatenation of different layers is always a Layer object that will be used as input for the training function.
For this purpose, I overloaded the `cat`, `horzcat` and `vertcat` operators for `Layers` and `subsref` for `Layer`. I also changed the `disp` method of these classes in order to properly print the layers details. `InputParser` is used in every class for the parameter management and the attributes setting. Corresponding code can be found in Section 4.4.1.

3.2.2 TrainNetwork

TrainNetwork is the function for performing the main function of the network: the training. The steps to be performed are:

1. Import the python module `TFinterface`
2. Instantiate a `TrainNetwork` object
3. For each layer, create a py object and pass it to the network
4. Compile the network: build the TF graph on the desired device
5. Train the network: run the TF session
6. Create a SeriesNetwork object

The user calls the training doing:

```python
net = trainNetwork(XTrain, TTrain, layers, options)
```
3.2.3 SeriesNetwork

Once the network is trained, the user can perform the prediction with the functions predict or classify:

```python
prediction, scores = classify(net, XTest)
prediction = predict(net, XTest)
```

3.3 Python

The layers implemented in python inherit from an abstract class Layer. The class Dataset handles the data input and all other functions are in the class TrainNetwork. The Octave TrainNetwork calls the python method add_layer via pytave passing a python dict with the information of the layer. The layers are added to the TF graph in build_graph called in compile. In method train the variables are initialized and the session run. Input nd-arrays are reshaped due to a bug of Pytave (see Section 4.9).

4 The code

4.1 Code repository

All the code implemented during the GSoC can be found in the repository:

https://bitbucket.org/cittiberto/gsoc-octave-nnet/commits/all

(username: cittiberto, bookmark enrico)

Since I implemented a completely new part of the package, I pushed the entire project in three commits and need some other commit for providing a PR to the GNU Octave community [11].

4.2 Commits

1. The first commit (ade115a) contains the layers. There is a class for each layer, with a corresponding function which calls the constructor. All the layers inherit from a Layer class which lets the user create a layers concatenation, that is the network architecture. Layers have several parameters, for which I have guaranteed the compatibility with Matlab [12].

2. The second commit (479ecc5) is about the Python part, including an init file checking the Tensorflow installation. I implemented a Python module, TFInterface, which includes:

   layer.py: an abstract class for layers inheritance
   layers/layers.py: the layer classes that are used to add the right layer to the TF graph
   dataset.py: a class for managing the datasets
   input trainNetwork.py: the core class, which initiates the graph and the session, performs the training and the predictions
   deepdream.py: a version of [13] for deepdream implementation (it has to be completed)
3. The third commit (e7201d8) includes:

**trainingOptions**: All the options for the training. Up to now, the only optimizer available is the stochastic gradient descent with momentum (sgdm) implemented in the class TrainingOptionsSGDM. **trainNetwork**: passing the data, the architecture and the options, this function performs the training and returns a SeriesNetwork object **SeriesNetwork**: class that contains the trained network, including the Tensorflow graph and session. This has three methods: predict: predicting scores for regression problems classify: predicting labels for classification problems activations: getting the output of a specific layer of the architecture

### 4.3 Testing Tensorflow installation

I’ve spent some time wondering which was the best solution to test the correct installation of the Tensorflow Python APIs on the machine. The last solution was putting the test in a function `__nnet_init__` and adding it in the PKG_ADD in order to be executed when the package is loaded.

```matlab
function subdir_paths = __nnet_init__ ()

    ## Check if Pytave is installed
    try
        py;
        catch
            error ("This package needs Pytave. \n", ...) |
            "Download it at https://bitbucket.org/mtmiller/pytave");
        end_try_catch

    ## Check if Python is installed
    try
        [~, ~, v] = pyversion ();
        assert (v, true);
        catch
            error ("This package needs Python.");
        end_try_catch

    ## Check if Tensorflow is installed
    try
        py.tensorflow.VERSION;
        catch
            error ("This package needs Tensorflow as backend engine. Please ", ...) |
            "make sure you have the Python TensorFlow API installed.");
        end_try_catch

endfunction
```

Listing 1: Testing TF installation
4.4 Layers in Octave

4.4.1 Layer class

trainNetwork takes as input the data and two objects: the layers and the options. I decided to store the layers (of type Layers) in a cell array as attribute of the Layer class. Overloading the subsref, I let the user call a specific layer with the ‘()’ access, like a classic array. With this kind of overloading I managed to solve the main problem of this structure, that is the possibility to get a property of a layer doing for example layers(1).Name

```matlab
classdef Layer < handle
    properties (Access = private) layers = {}; endproperties
    methods (Hidden, Access = {? Layers})
        ## constructor: store layers in attribute
        function this = Layer (varargin)
            nargin = numel (varargin);
            this.layers = cell(1, nargin);
            for i = 1:nargin
                this.layers{i} = varargin{i};
            end
        endfunction
    endmethods
    methods (Hidden)
        ## overload of subsref for accessing class object.
        function obj = subsref (this, idx) switch idx(1).type
            case '()'
                idx(1).type = '{}';
                obj = builtin ('subsref', this.layers, idx);
            case '{}'
                error ('{} indexing not supported');
            case '.'
                obj = builtin ('subsref', this, idx);
            endswitch
        endfunction
        function obj = numel (this)
            obj = builtin ('numel', this.layers);
        endfunction
        function obj = size (this)
            obj = builtin ('size', this.layers);
        endfunction
    endmethods
endclassdef
```

Listing 2: Layer class in GNU Octave
### 4.4.2 Layers class

Base layer class

```octave
classdef Layers < handle
    properties (SetAccess = protected, Abstract = true)
        Name = "";
    endproperties

    methods
        ## use inputParser for managing the inputs
        function this = Layers (varargin)
            p = inputParser ();
            p.KeepUnmatched = true;
            p.FunctionName = "Layers";
            addOptional (p, "par1", 0);
            addOptional (p, "par2", 0);
            addParameter (p, "Name", ",", @ischar);
            parse (p, varargin{:});
            this.Name = p.Results.Name;
        endfunction

        ## methods for printing
        function index_name (this, index)
            this.Name = strcat (this.Name, "_", num2str (index));
        endfunction

        function set_default_name (this)
            name = this.default_name;
            this.Name = name;
        endfunction

        function disp (this)
            if ( nargin != 1)
                print_usage ();
            endif
            printf ('Layers object with attributes:

');
            printf (['\n  Name : %s
'], this.Name);
        endfunction
    endmethods

    ## these methods cannot be redefined in a subclass.
    methods (Sealed)
        ## overload of concatenating
        function obj = cat (obj1, varargin)
            obj = Layer (obj1, varargin{:});
        endfunction

        function obj = horzcat (obj1, varargin)
            obj = Layer (obj1, varargin{:});
        endfunction

        function obj = vertcat (obj1, varargin)
            obj = Layer (obj1, varargin{:});
        endfunction
    endmethods
endclassdef
```

Listing 3: Layers class in GNU Octave
An example of derived layer: DropoutLayer

```matlab
classdef DropoutLayer < Layers
    properties (SetAccess = protected)
        Probability = 0.5;
        default_name = "dropout";
    endproperties
    methods
        function this = DropoutLayer (varargin)
            this = this.Layers (varargin{:});
            p = inputParser ();
            p.FunctionName = "DropoutLayer";
            addOptional (p, "Probability", 0.5, @(x) any (x>=0 && x<=1 && isnumeric (x)));
            addParameter (p, "Name","", @ischar);
            parse (p, varargin{:});
            this.Probability = p.Results.Probability;
        endfunction
        function disp (this)
            if ( nargin != 1)
                print_usage ();
            endif
            printf (' DropoutLayer object with attributes:

');
            printf ('
    Name : "%s"

    Probability : %d

', this.Name , this.Probability 
);
    endfunction
    endmethods
endclassdef
```

Listing 4: Example of a layer in GNU Octave

The objects of these classes can be instantiated with a corresponding function, implemented in the directory inst/. Here an example for creating a Layer object

```matlab
> # TEST LAYERS
> a = imageInputLayer([2,2,3]); first layer
> b = convolution2dLayer(1,1); second layer
> c = dropoutLayer(1); third layer
> layers = [a b c]; Layer object from layers concat
> drop = layers(3); Layer element access
> drop.Probability Access layer attribute
ans = 0.50000
```

Listing 5: Testing layers in GNU Octave
4.5 Layers in Python

Base layer class:

```python
class Layer(ABC):
    def __init__(self, name, tensor_name):
        """init layer object
        :param name: custom name given by the user
        :param tensor_name: generated tensor name used in the activations
        function to retrieve layer output
        """
        self.name = name
        self.tensor_name = tensor_name

    @abstractmethod
    def call(self, **kwargs):
        pass
```

Listing 6: Layer class in Python

An example of derived layer: DropoutLayer

```python
class _DropoutLayer(Layer):
    """Dropout layer
    # Properties
    Probability: Dropout probability
    # Methods
    call: add the corresponding TF layer to the graph
    """
    def __init__(self, Probability, name):
        super(_DropoutLayer, self).__init__(name,
            'dropout/{}/Identity:0'.format(name))
        self.Probability = Probability

    def call(self, inputs):
        with tf.name_scope('dropout'):
            return tf.layers.dropout(
                inputs,
                rate=self.Probability,
                noise_shape=None,
                seed=None,
                training=False,
                name=self.name
            )
```

Listing 7: Example of a layer in Python
4.6 TrainingNetwork in Python

Here the main methods of TrainingNetwork. For the details of the class definition, see the documentation doc/TFinterface/index.html in the repository.

- handle_data: create dataset class for managing the batches
- handle_input: this method checks if the first layer is an input layer and raise an error otherwise
- check_architecture: it checks the consistency of the architecture. It raises an error if there is a problem in the output or in layers order.
- add_layer: add a layer to self.layers. The layer is passed from Octave as a Python Dict
- compile: the TF graph is built choosing the desired device (cpu or gpu)
- build_graph: all layers of self.layers are added to the graph
- loss_def: define the loss function. Possibilities are mean squared error for the regression and categorical crossentropy for classification
- optim_def: define the optimizer: sgd with momentum. Exponential decay of the learning rate is allowed
- train: main function of training. The steps are:
  - Create Dataset object
  - Define batches number
  - Get TF tensors from the graph
  - Initialize variables and session
  - Define decay steps for learning rate
  - Loop over iterations: run the graph, print the results and if required, save checkpoint
- def_checkpoints: save the checkpoints with the network parameters
- classify: predict the label for new images with the score
- predict: predict the label for new images
- prediction: compute the prediction for predict and classify
- activations: get output of a specific layer
4.7 Brief tutorial

1. Install Python Tensorflow API (as explained in [15])
2. Install Pytave (following these instructions [14])
3. Install nnet package (In Octave: install [16] and load [17])
4. Check the package with
   \[
   \text{make check PYTAVE="pytave/dir/"}
   \]

Listing 8: bash version

5. Open Octave, add the Pytave dir the the paths and run your first network:

### TRAINING ###
# Load the training set
[XTrain,TTrain] = digitTrain4DArrayData();

# Define the layers
layers = [imageInputLayer([28 28 1]);
          convolution2dLayer(5,20);
          reluLayer();
          maxPooling2dLayer(2,'Stride',2);
          fullyConnectedLayer(10);
          softmaxLayer();
          classificationLayer()];

# Define the training options
options = trainingOptions('sgdm', 'MaxEpochs', 15, 'InitialLearnRate', 0.04);

# Train the network
net = trainNetwork(XTrain,TTrain,layers,options);

### TESTING ###
# Load the testing set
[XTest,TTest]= digitTest4DArrayData();

# Predict the new labels
YTestPred = classify(net,XTest);

Listing 9: Example for image classification

4.8 Example: CNN for regression

Here I chose an interesting example proposed in the Matlab examples at [18]. The dataset is composed by 5000 images, rotated by an angle $\alpha$, and a corresponding integer label (the rotation angle $\alpha$). The goal is to make a regression to predict the angle of a rotated image and straighten it up.

I have kept the structure as in the Matlab example, but I generated a new dataset starting from LeCun’s MNIST digits (datasets at [19]). Each image was rotated by
a random angle between $-70$ degrees and 70 degrees, in order to keep the right orientation of the digits (code in dataset_generation.m). In Fig. 7 some rotated digits with the corresponding original digits.

The implemented linear model is:

$$\hat{Y} = \omega X + b,$$

where the weights $\omega$ and the bias $b$ will be optimized during the training minimizing a loss function. As loss function, I used the mean square error (MSE):

$$\frac{1}{n} \sum_{i=1}^{n} (\hat{Y}_i - Y_i)^2,$$

where the $Y_i$ are the training labels.

In order to show the effective improvement given by a Neural Network, I started to make a simple regression feeding the $X$ variable of the model directly with the $28 \times 28$ images. Even if for the MSE minimization a close form exists, I implemented an iterative method for discovering some Tensorflow features (code in regression.py). For evaluate the accuracy of the regression, I consider a correct regression if the difference between angles is less than 20 degrees. After 20 epochs, the convergence was almost reached, giving an accuracy of 0.6146.

I want to analyze now the improvement given by a feature extraction performed with a convolutional neural network (CNN). As in the Matlab example, I used a basic CNN since the input images are quite simple (only numbers with monochromatic background) and consequently the features to extract are few. Schema of the model in Fig. 8.

**INPUT [28x28x1]** will hold the raw pixel values of the image, in this case an image of width 28, height 28

**CONV** layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region
they are connected to in the input volume. This results in volume such as \([12\times12\times25]\): 25 filters of size 12x12

\textbf{RELU} layer will apply an element-wise activation function, such as the \(\max(0, x)\) thresholding at zero. This leaves the size of the volume unchanged (\([12\times12\times25]\)).

\textbf{FC} (i.e. fully-connected) layer will compute the class scores, resulting in volume of size \([1\times1\times1]\), which corresponds to the rotation angle. As with ordinary Neural Networks, each neuron in this layer will be connected to all the numbers in the previous volume.

We can visualize the architecture with Tensorboard where the graph of the model is represented (Fig. 9).

The results are quite satisfying: it reached an accuracy of 0.75 (205 seconds overall). One can see loss and accuracy in Fig. 10 and in Fig. 11 the marked improvement of the regression.

With the same parameters, Matlab reached an accuracy of 0.76 in 370 seconds (code in \texttt{regression\_Matlab\_nnet.m}), so the performances are good.

\section{4.9 Troubleshooting}

\textbf{JAVA\_HOME} Compiling Octave by source, a common error is "WARNING: JAVA\_HOME environment variable not initialized.". A useful fix can be found in this blog.

\textbf{Pytave: nd-arrays} Pytave has some problems to pass nd-arrays from Octave to Python and viceversa. As workaround, just flatten the array before and after the passage through Pytave.

\textbf{Pytave: import Tensorflow} Importing Tensorflow via Pytave with
\begin{verbatim}
py.tensorflow.VERSION
\end{verbatim}
one can have one of these errors:
Figure 9: Model graph generated with Tensorboard

Figure 10: Loss and accuracy for training and testing

Figure 11: Rotated images in columns 1,3,5 and after the CNN regression in columns 2,4,6
pycall: AttributeError: 'module' object has no attribute 'argv'
error: pycall: ImportError: cannot import name pywrap tensorflow
Be sure to have at least Tensorflow >= 1.0.1, Octave >= 4.2.0 and Pytave changeset >= 14b134f.

Make check with pytave In order to compute the make check, be careful to pass the Pytave path. This in the right command:
make check PYTAVE="/dir/to/pytave/"

Tests fail in make check During the make check, 6 tests fail because of 6 features not yet implemented:

- inst/convolution2dLayer.m ................................... PASS 4/6
- inst/crossChannelNormalizationLayer.m .............. PASS 4/5
- inst/deepDreamImage.m ....................................... PASS 1/2
- inst/fullyConnectedLayer.m .................................. PASS 4/6

5 Future improvements

Missing features I did not manage to implement some features because of the lack of the TF functions. These features will be probably implemented soon and they could be integrated in the package.

<table>
<thead>
<tr>
<th>Function</th>
<th>Missing features</th>
</tr>
</thead>
<tbody>
<tr>
<td>activations</td>
<td>OutputAs (for changing output format)</td>
</tr>
<tr>
<td>imageInputLayer</td>
<td>DataAugmentation and Normalization</td>
</tr>
<tr>
<td>trainNetwork</td>
<td>Accepted inputs: imds or tbl</td>
</tr>
<tr>
<td>trainNetwork</td>
<td>.mat Checkpoints</td>
</tr>
<tr>
<td>trainNetwork</td>
<td>ExecutionEnvironment: 'multi-gpu' and 'parallel'</td>
</tr>
<tr>
<td>TrainingOptions</td>
<td>WorkerLoad and OutputFcn</td>
</tr>
<tr>
<td>DeepDreamImages</td>
<td>Generalization to any network and AlexNet example</td>
</tr>
</tbody>
</table>

Future improvements Main improvements could be:

- Manage the session saving
- Save the checkpoints as .mat files and not as TF checkpoints
- Optimize array passage via Pytave
- Categorical variables for classification problems

Furthermore, the package could be integrated with Recurrent Neural networks (RNNs), applied in domains where features depend on the sequence (e.g. natural language).
References


[14] https://bitbucket.org/mtmiller/pytave

[15] https://www.tensorflow.org/install/


