# TNNF: Tiny Neural Network Framework

Data Science Group
@ SoftServe Inc.

Roman Grubnyk, Ihor Kostiuk, Iurii Milovanov and Chris Poulin
{rhrub,ikostiuk,imilov,cpoulin} @ softserveinc.com

**Abstract**

TNNF is a Deep Learning research library developed by Data Science Group at SoftServe Inc. In this article we give a brief history of the library and an overview of its basics and architecture. Also we provide step-by-step instructions on how to get TNNF running and give some examples on how to use it.

## 1 Introduction

### 1.1 Motivation

Data Science Group (DSG) was founded at SoftServe in 2013 as a highly innovative research and engineering lab. Since then we have been working internally on the scientific library that we have successfully used in our research. Our goal was to develop flexible and 'user friendly' solution that would be easy to extend and integrate into production or existing solution. Also it was a great opportunity for us to develop our expertise in Deep Learning by implementing recent state-of-the-art algorithms and techniques. Now we have published it open source in order to share our experience and to engage new contributors into this project.

### 1.2 Why Python?

TNNF is written in Python. Obviously Python is very efficient, easy-to-read and powerful high-level programming language. But what is more important, it has a large and comprehensive standard library and a wide array of third-party extensions for different purposes (such as math, image and text processing, databases etc) which make it easier to prototype, develop and test complex solutions in single environment.

## 2 Overview

In this section we give a brief and high-level overview of TNNF library and provide some information that you might find useful before you start.

### 2.1 Theano

TNNF is built on top of Theano library. Particularly, it wraps Theano code and provides user with easy-to-use API interface which makes building neural network architecture simple. At the same time, the library was built to be modular.

### 2.2 Features

**List of features available in TNNF:**

- Feedforward neural networks:
  - Multilayer perceptron (MLP)
  - Convolutional neural network (CNN)
  - Dropout
- Recurrent neural networks:
  - Long Short Term Memory (LTSM)
- Optimization algorithms:
  - Gradient Descent (GD)
  - Stochastic Gradient Descent (SGD)
  - Rprop
  - RMSProp
- Generalization features:
  - Sparsity constraint
  - Weight decay
- Activation functions:
  - Linear
  - Sigmoid
  - Tanh
  - Softmax
  - Maxout
  - ReLU/LReLU
- GPU-accelerated computing (using Theano with CUDA backend)
- Model persistence (serialization)
- Debug and visualization tools

### 2.3 File Listing

**TNNF library consists of the following files:**

- *fTheanoNNclassCORE.py* – main TNNF module;
- *fGraphBuilderCORE.py* – methods for building graphics and plots, such as Learning Curves, ROCs etc;
- *fDataWorkerCORE.py* – data manipulation methods;
- *fCutClassCORE.py, fImageWorkerCORE.py* – methods for working with images, such as segmentation, preprocessing etc;

# 3 Getting Started

## 3.1 Requirements

**Note**

Please check the following system requirements before you start an installation.

### 3.1.1 Operating Systems

Here is the list of Operating Systems supported by TNNF:

- Linux
- Mac OS X
- Windows

Note: Only 64-bit Linux architecture is well-tested;

### 3.1.2 Python version

Make sure that you have an appropriate Python version (2.7 or higher) installed. Check it by typing the following in command line:

```
user@host:~$ python --version
Python 2.7.5
```

Update your Python version if necessary;

### 3.1.3 Python dependencies

Here is the list of libraries that TNNF depends on:

- **Theano** – core component;
- **NumPy** – high-level mathematical framework
- **PIL** – image processing library;
- **matplotlib** – plotting framework;
- **cPickle** – basic serialization tool;
- **h5py** – tool to store and organize large amounts of numerical data;

Install additional Python libraries if necessary:

- Ubuntu/Debian:

```
user@host:~$ sudo apt-get install python-pil python-numpy python-scipy python-dev python-pip
 ↪   python-nose g++ libopenblas-dev git
user@host:~$ sudo pip install Theano
```

- (RedHat/CentOS) Register the following extra software repositories:

```
user@host:~$ sudo yum install
 ↪   http://dl.fedoraproject.org/pub/epel/7/x86_64/e/epel-release-7-2.noarch.rpm
user@host:~$ sudo yum install
 ↪   http://pkgs.repoforge.org/rpmforge-release/rpmforge-release-0.5.3-1.el7.rf.x86_64.rpm
```

- (RedHat/CentOS) Install an appropriate packages:

```
user@host:~$ sudo yum install python-devel python-nose python-pillow python-setuptools gcc
 ↪   gcc-gfortran gcc-c++ blas-devel lapack-devel atlas-devel
user@host:~$ sudo easy_install pip
user@host:~$ sudo pip install numpy==1.6.1
user@host:~$ sudo pip install scipy==0.10.1
user@host:~$ sudo pip install Theano
```

- For other systems we highly recommend to follow the installation instructions provided by each particular library;

#### 3.1.4 GPU computing

Please refer to Theano documentation page [1] for more information on how to install and configure CUDA-backend in case if your GPU device is CUDA-ready.

### 3.2 Installation

#### 3.2.1 Downloading

1. Get the latest TNNF version from GitHub:

   ```
   user@host:~$ git clone https://github.com/spaceuniverse/TNNF
   ```

#### 3.2.2 Test the installation

1. Navigate to 'CORE' folder in project's root directory:

   ```
   user@host:~$ cd  TNNF/CORE/
   ```

[1] http://deeplearning.net/software/theano/tutorial/using_gpu.html

2. Run the simple test by typing the following in command line:

   ```
   user@host:~$ python -c "import fTheanoNNclassCORE"
   ```

   It should succeed without error;

3. Continue with Examples section;

#### 3.2.3 Updating

1. Navigate to 'CORE' folder in project's root directory:

   ```
   user@host:~$ cd  TNNF/CORE/
   ```

2. Run the following command in order to check TNNF for updates:

   ```
   user@host:~$ git pull
   ```

## 4 Benchmarks

In this section we provide the comparison of TNNF's precision and run-time performance with that of other Deep Learning libraries.

### 4.1 Pylearn2

TBA

### 4.2 Torch7

TBA

## 5 Examples

### 5.1 Classification with Neural Networks

We will try do describe how to use TNNF to solve a simplest artificial task we were able to design.

Let's create a simple classifier that will assign 0 or 1 class to data from input. And as we don't want random classification, let's try to train our classifier on previously manually labeled data.

#### 5.1.1 Data

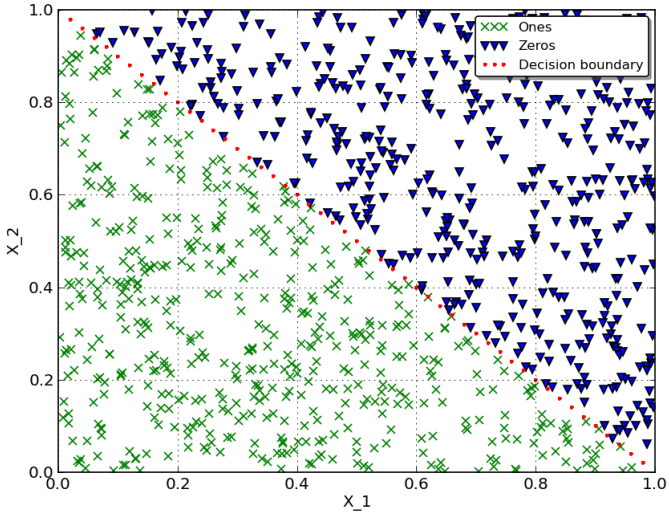
Let's imagine we have some amount of manually labeled data (to label data we will use particular "decision rule"). This data will be used to train our classifier.

In this particular task we assume we have to classify a pair of numbers $(X_1, X_2)$,

where$X_1, X_2 \in [0, 1)$ and assign each pair 0 or 1 class. To manually label this data let's use such formula:

$$Label = \begin{cases} 0 & \text{if: } -X_1 + 1 < X_2 \\ 1 & \text{if: } -X_1 + 1 \geq X_2 \end{cases} \tag{1}$$

Here is how our labeled data looks on 2D plot:




### 5.1.2 Neural Network

As was mentioned - let's use TNNF to solve this task.

What we have:

- Randomly generated data for training
- Randomly generated data for cross-validation
- Labels

What we want to achieve:

- Given generated data - using TNNF - predict correct labels

To do this, we will use one-layer Neural Network with simplest Linear activation function and architecture:

- Input layer: 2 neurons
- Output layer: 1 neuron

To define **predicted** label we will round the activation of output layer:

$$Output = \begin{cases} 0 & activation \leq 0.5 \\ 1 & activation > 0.5 \end{cases} \tag{2}$$

Here is how it looks in code.

Set general options for whole network, such as:

- Learning step
- Size of mini-batch we will use (in this case we use full batch)
- Size of cross-validation set

```
#Common options for whole NN
options = OptionsStore(learnStep=0.05,
                       minibatch_size=dataSize,
                       CV_size=dataSize)
```

Describe per layer architcture. Set:

- Number of neurons on layer's input
- Number of neurons on layer's output
- Specify activation function to use

```
#Layer architecture
#We will use only one layer with 2 neurons on input and 1 on
↪  output
L1 = LayerNN(size_in=dataFeatures,
             size_out=1,
             activation=FunctionModel.Linear)
```

Put everything together and create network object:

```
#Compile NN
NN = TheanoNNclass(options, (L1, ))
```

### 5.1.3 How it performs

Instead of talking how well it performs its better to show this. To visualise predicted boundary we will use networkâĂŹs weights and bias that it learned during training.

As we use Linear activation function, we can write a formula to calc output activation:

$$activation = w_1 X_1 + w_2 X_2 + b \tag{3}$$

Using our previous formula to define predicted label, we can rewrite this as follows:

$$w_1 X_1 + w_2 X_2 + b \gtrless 0.5 \tag{4}$$

To be able to draw predicted decision boundary we need to express $X_2$:

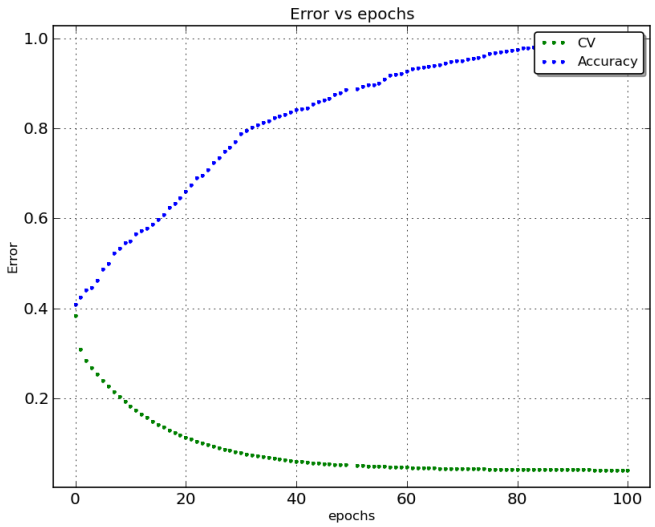
$$X_2 \gtrless -\frac{w_1}{w_2} X_1 + \frac{0.5 - b}{w_2} \tag{5}$$

Here is how it looks like in code:

```
#Recalculate predicted decision boundary
y_predicted = -w1 * x / w2 + (0.5 - b) / w2
```

If you enable drawing and set reasonable drawEveryStep you will get number of pictures that shows how Neural Net evolves.

Here we can track how accuracy and network error evolves vs iterations:




It almost reach 100% accuracy!

### 5.1.4 Full script listing

```
import numpy as np
import unittest
import os
import sys
sys.path.append('../../../CORE')
from fTheanoNNclassCORE import OptionsStore, TheanoNNclass,
↪ NNsupport, FunctionModel, LayerNN
from fDataWorkerCORE import csvDataLoader
from fGraphBuilderCORE import Graph
from matplotlib.pylab import plot, title, xlabel, ylabel, legend,
↪ grid, margins, savefig, close, xlim, ylim

dataSize = 1000
dataFeatures = 2

#Supposed boundary line
#Where x - [0] row, f(x) - [1] row in data
# if f(x) < -x + 1 - then label = 1
#                    else label = 0

#Create random data [0, 1)
data = np.random.rand(dataFeatures, dataSize)

#Create random cross-validation [0, 1)
CV = np.random.rand(dataFeatures, dataSize)

#Create array for labels
labels = np.zeros((1, dataSize))

#Create array for cross-validation labels
CV_labels = np.zeros((1, dataSize))

#Calc labels (and cross-validation) based on supposed boundary
↪ decision function analytically
labels[0, :] = -data[0, :] + 1 > data[1, :]
CV_labels[0, :] = -CV[0, :] + 1 > CV[1, :]

#Let's draw our data and decision boundary we use to divide it
#Calc decision boundary
x = np.arange(0, 1.0, 0.02)
y = -x + 1

#Draw labeled data
#Uncomment next part if you want to visualise input data
'''
ones = np.array([[], []])
```

```
zeros = np.array([[], []])
for i in xrange(dataSize):
    if labels[0, i] == 0:
        zeros = np.append(zeros, data[:, i].reshape(-1, 1),
↪  axis=1)
    else:
        ones = np.append(ones, data[:, i].reshape(-1, 1),
↪  axis=1)

xlim(0, 1)
ylim(0, 1)

plot(ones[0, :], ones[1, :], 'gx', markeredgewidth=1,
↪  label='Ones')
plot(zeros[0, :], zeros[1, :], 'bv', markeredgewidth=1,
↪  label='Zeros')
plot(x, y, 'r.', markeredgewidth=0, label='Decision boundary')
xlabel('X_1')
ylabel('X_2')

legend(loc='upper right', fontsize=10, numpoints=3, shadow=True,
↪  fancybox=True)
grid()
savefig('data_visualisation.png', dpi=120)
close()
'''

#Check average
avgLabel = np.average(labels)

print data.shape
print 'Data:\n', data[:, :6]
print labels.shape
print 'Average label (should be ~ 0.5):', avgLabel

#For now we have labeled and checked data.
#Let's try to train NN to see, how it solves such task
#NN part

#Common options for whole NN
options = OptionsStore(learnStep=0.05,
                       minibatch_size=dataSize,
                       CV_size=dataSize)

#Layer architecture
#We will use only one layer with 2 neurons on input and 1 on
↪  output
L1 = LayerNN(size_in=dataFeatures,
```

```
            size_out=1,
            activation=FunctionModel.Linear)

#Compile NN
NN = TheanoNNclass(options, (L1, ))

#Compile NN train
NN.trainCompile()

#Compile NN oredict
NN.predictCompile()

#Number of iterations (cycles of training)
iterations = 1000

#Set step to draw
drawEveryStep = 100
draw = False

#CV error accumulator (for network estimation)
cv_err = []

#Accuracy accumulator (for network estimation)
acc = []

#Main cycle
for i in xrange(iterations):

    #Train NN using given data and labels
    NN.trainCalc(data, labels, iteration=1, debug=True)

    #Draw data, original and current decision boundary every
    ↪ drawEveryStep's step
    if draw and i % drawEveryStep == 0:

        #Get current coefficient for our network
        b = NN.varWeights[0]['b'].get_value()[0]
        w1 = NN.varWeights[0]['w'].get_value()[0][0]
        w2 = NN.varWeights[0]['w'].get_value()[0][1]

        #Recalculate predicted decision boundary
        y_predicted = -w1 * x / w2 + (0.5 - b) / w2

        #Limit our plot by axes
        xlim(0, 1)
        ylim(0, 1)

        #Plot predicted decision boundary
```

```
        plot(x, y_predicted, 'g.', markeredgewidth=0,
        ↪  label='Predicted boundary')

        #Plot original decision boundary
        plot(x, y, 'r.', markeredgewidth=0, label='Original
        ↪  boundary')

        #Plot raw data
        plot(data[0, :], data[1, :], 'b,', label='data')

        #Draw legend
        legend(loc='upper right', fontsize=10, numpoints=3,
        ↪  shadow=True, fancybox=True)

        #Eanble grid
        grid()

        #Save plot to file
        savefig('data' + str(i) + '.png', dpi=120)

        #Close and clear current plot
        close()

        #Estimate Neural Network error (square error, "distance"
        ↪  between real and predicted value) on
        ↪  cross-validation
        cv_err.append(NNsupport.crossV(CV_labels, CV, NN))

        #Estimate network's accuracy
        accuracy = np.mean(CV_labels == np.round(NN.out))
        acc.append(accuracy)

        #Draw how error and accuracy evolves vs iterations
        Graph.Builder(name='NN_error.png', error=NN.errorArray,
        ↪  cv=cv_err, accuracy=acc, legend_on=True)
```

### 5.2 Sparse Autoencoder

TBA