ΟΙΚΟΝΟΜΙΚΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΑΘΗΝΩΝ

ATHENS UNIVERSITY
OF ECONOMICS
AND BUSINESS

Department of Informatics
MsC in Computer Science

"...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.

Dr. Robert Hecht-Nielsen
In "Neural Network Primer: Part I" by Maureen Caudill, AI Expert, Feb. 1989

Simple Neural Network

Project Report

Members Mitsopoulou Eleni
Vitsas Nikolaos

Supervisor Titsias Michail

Athens, June 2017

Abstract

*In this work we present an implementation of a **Simple Neural Network (SNN)** to perform multi-classification and prediction. The SNN use the Cross-entropy error function across with a Softmax activation function for the output neurons. The hidden layer neurons can use among three different activations functions, namely Smooth Rectifier, Hyperbolic tangent and Cosine.*

1 Introduction

In recent days Artificial Neural Networks (ANNs) are becoming more and more popular. The basic idea behind them is to simulate (one could say copy in a simplified but reasonably faithful way) lots of densely interconnected brain cells inside a computer in order to effectively learn things, recognize patterns, and make decisions in a humanlike way. The important thing about an ANN is that you don't have to program it to learn explicitly: it learns all by itself, just like a brain, but it isn't a brain.

ANNs are typically organized in layers. Layers are made up of a number of interconnected 'nodes' which contain an 'activation function'. Patterns are presented to the network via the 'input layer', which communicates to one or more 'hidden layers' where the actual processing is done via a system of weighted 'connections'. The hidden layers then link to an 'output layer' where the answer comes from the output/s as shown in the Figure 1.

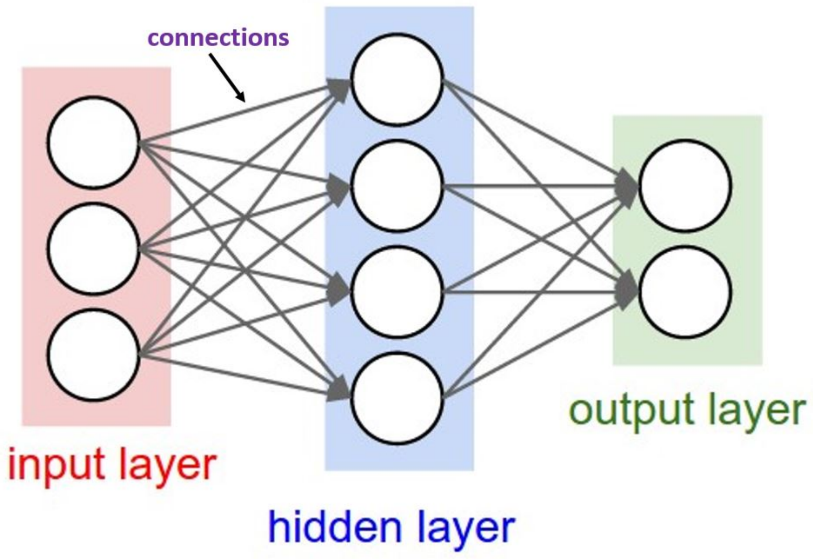


Figure 1: Example of an Artificial Neural Network.

Most ANNs contain some form of 'learning rule' which modifies the weights of the connections according to the input patterns that it is presented with. There are many different kinds of learning rules used by neural networks. This work is concerned only with one; the chain rule. The chain rule is often utilized by the most common class of ANNs called 'backpropagational neural networks' (BPNNs). Backpropagation is an abbreviation for the backwards propagation of error.

With the chain rule, as with other types of backpropagation, 'learning' is a supervised process that occurs with each cycle or 'epoch' (i.e. each time the network is presented with a new input pattern) through a forward activation flow of outputs, and the backwards error propagation of weight adjustments. More simply, when a neural network is initially presented with a pattern it makes a random ‘guess’ as to what it might be. It then sees how far its answer was from the actual one and makes an appropriate adjustment to its connection weights. Within each hidden layer node is an activation function which polarizes network activity and helps it to stabilize.

Backpropagation performs a gradient descent (or ascent, depending on error’s function sign) within the solution’s vector space towards a ‘global minimum’ along the steepest vector of the error surface. The global minimum is that theoretical solution with the lowest possible error. The error surface itself is a hyper-paraboloid but is seldom ‘smooth’. In most cases, the solution space is quite irregular with numerous ‘pits’ and ‘hills’ which may cause the network to settle down in a ‘local minimum’ which is not the best overall solution.

Since the nature of the error space can not be known a priori, neural network analysis often requires a large number of individual runs to determine the best solution. Most learning rules have built-in mathematical terms to assist in this process which control the ‘speed’ (Beta-coefficient) and the ‘momentum’ of the learning. The speed of learning is actually the rate of convergence between the current solution and the global minimum. Momentum helps the network to overcome obstacles (local minima) in the error surface and settle down at or near the global minimum.

Once a neural network is ‘trained’ to a satisfactory level it may be used as an analytical tool on other data. To do this, the user no longer specifies any training runs and instead allows the network to work in forward propagation mode only. New inputs are presented to the input pattern where they filter into and are processed by the middle layers as though training were taking place, however, at this point the output is retained and no backpropagation occurs. The output of a forward propagation run is the predicted model for the data which can then be used for further analysis and interpretation.

The rest of this document, is structured as follows. In Section 2, we discuss and prove several of the mathematical methods that are used to perform multi-label classification. In Section 3 we cover the most important implementation details, and finally in Section 4 we present experimental results with varying parameters.

2 Background

2.1 Error and Activation Functions

In this work we implement a Simple Neural Network (SNN), with one hidden layer, that performs multi-label classification. The error function used in our multi-label classification neural network is the **Cross-entropy**, and is given by the following equation:

$$E(\mathbf{w}) = \sum_{n=1}^{N}\sum_{k=1}^{K} t_{nk} log y_{nk} - \frac{\lambda}{2}||\mathbf{w}||^2 \tag{1}$$

where y_{nk} is the activation function of the output neurons, in our case **Softmax**, and is given by:

$$y_{nk} = \frac{e^{(\mathbf{w}_k^{(2)})^T \mathbf{z}_n}}{\sum_{j=1}^{K} e^{(\mathbf{w}_j^{(2)})^T \mathbf{z}_n}} \tag{2}$$

The hidden layer neurons can use one among three different activations functions namely **Smooth Rectifier**, **Hyperbolic tangent** and **Cosine**, and are given by the following equations respectively:

$$h(a) = log(1 + e^a) \tag{3}$$

$$h(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \tag{4}$$

$$h(a) = cos(a) \tag{5}$$

2.2 Notations

We will denote and use the following notations:

- The subscript k denotes the output layer.
- The subscript j denotes the hidden layer.
- The subscript i denotes the input layer.
- w_{kj} denotes a weight from the hidden to the output layer.
- w_{ji} denotes a weight from the input to the hidden layer.
- w_{j0}, w_{k0} denote the bias values.
- t denotes a target value.
- x denotes an input value.
- h denotes an activation function.
- out denotes an activation value.
- net denotes the net input (weighted sum).

2.3 The Forward Pass

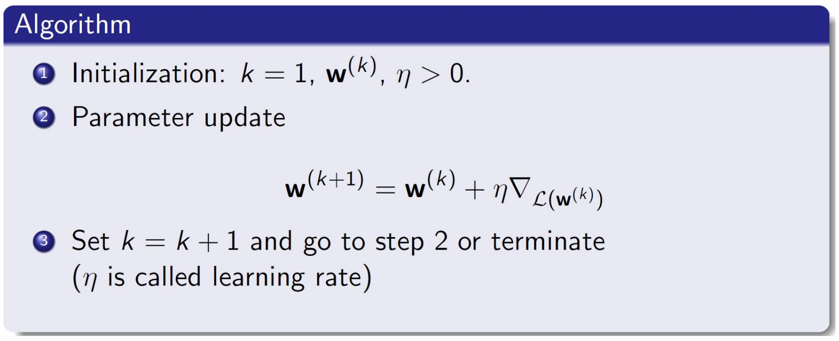
The Forward Pass is given by the following equation:

$$y_k(\mathbf{x},\mathbf{w}) = h_k\left(\sum_{j=1}^{M} w_{kj}^{(2)} h\left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{6}$$

where h_k represents the outer activation function, which is **Softmax** in our case, and h represents the hidden layer's activation function that could be one of the **Smooth Rectifier**, **Hyperbolic tangent** and **Cosine**.

2.4 The Backwards Pass

The Backwards Pass (i.e backpropagation) is used to update input and output weights that are stored respectively in the arrays $W^{(1)}$ & $W^{(2)}$ in the network, so that they cause the actual output to be closer with the target output. Thereby, minimizing the error for each output neuron and the network as a whole. The **gradient ascent** algorithm is used to estimate the proper step values(deltas) to update both $W^{(1)}$, $W^{(2)}$. The algorithm steps are shown in Figure 2.


Algorithm

1. Initialization: $k = 1$, $\mathbf{w}^{(k)}$, $\eta > 0$.
2. Parameter update

$$\mathbf{w}^{(k+1)} = \mathbf{w}^{(k)} + \eta \nabla_{\mathcal{L}(\mathbf{w}^{(k)})}$$

3. Set $k = k + 1$ and go to step 2 or terminate
(η is called learning rate)

Figure 2: Gradient descent/ascent.

2.4.1 Output Layer

Starting backpropagation from the Output Layer, the array $\nabla E(\mathbf{w}^{(2)})$ that is needed in order to update $W^{(2)}$ is given by:

$$\nabla E(\mathbf{w}^{(2)}) = (T - S)^T Z - \lambda W^{(2)} \tag{7}$$

where T is an array that stores the targeted data i.e $[T]_{nk} = t_{nk}$, S is an array that stores the probabilities calculated at the output layer after applying Softmax i.e $[S]_{nk} = y_{nk}$ and finally Z is an array that stores the outputs of the hidden layer neurons.

At this point, as we will need it for a later use, we prove that $(T - S)$ is derived from $\frac{\partial E(\mathbf{w})}{\partial net_k}$. For simplicity, we prove the case for a single feature vector. Here, K refers to the number of classes.

$$\frac{\partial E(\mathbf{w})}{\partial net_k} = \sum_{n=1}^{K} \frac{\partial t_n log(y_n)}{\partial net_k} = \sum_{n=1}^{K} t_n \frac{\partial log(y_n)}{\partial net_k} = \sum_{n=1}^{K} t_n \frac{1}{y_n} \frac{\partial y_n}{\partial net_k} \tag{8}$$

$$= \frac{t_k}{y_k} \frac{y_k}{net_k} + \sum_{n \neq k}^{K} \frac{t_n}{y_n} \frac{y_n}{net_k} = \frac{t_k}{y_k} y_k(1 - y_k) + \sum_{n \neq k}^{N} \frac{t_n}{y_n}(-y_n y_k) \tag{9}$$

$$= t_k - t_k y_k - \sum_{n \neq k}^{K} t_n y_k = t_k - \sum_{n=1}^{K} t_n y_k = t_k - y_k \sum_{n=1}^{K} t_n \tag{10}$$

$$= t_k - y_k \tag{11}$$

Note that we derived $\frac{\partial y_n}{\partial net_k}$ for $n = k$ and $n \neq k$ above which is the derivative of the **softmax**.

When considering the case of multiple feature vectors, the last equation can be written in matrix form, $(T - S)$ which is what we wanted to prove. This means that the update delta for $W^{(2)}$ can be easily calculated in the first phase of back propagation.

2.4.2 Hidden Layer

We now continue the backpropagation to the Hidden Layer. We need to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. Each output of a neuron in the hidden layer affects all of the nodes in the output layer. Therefore, when back-propagating the error, we need to calculate the cumulative effect of all deltas. The array $\nabla E(\mathbf{w}^{(1)})$ that is needed in order to update $W^{(1)}$ is calculated by applying the chain rule:

$$\nabla E(\mathbf{w}^{(1)}) = \frac{\partial E(\mathbf{w})}{\partial w_{ji}^{(1)}} = \left(\sum_{k=1}^{K} \frac{\partial E(\mathbf{w})}{\partial out_k} * \frac{\partial out_k}{\partial net_k} * \frac{\partial net_k}{\partial out_j} \right) * \frac{\partial out_j}{\partial net_j} * \frac{\partial net_j}{\partial w_{ji}^{(1)}} \tag{12}$$

The net input for each neuron j at the hidden layer is given by:

- $net_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}$

The output for each neuron j and for each of the three different activation functions that could be used at the hidden layer is given by:

- $out_j = log(1 + e^{net_j})$
- $out_j = \frac{e^{net_j} - e^{-net_j}}{e^{net_j} + e^{-net_j}}$
- $out_j = cos(net_j)$

The net input for each neuron k at the output layer is given by:

- $net_k = \sum_{j=1}^{M} w_{kj}^{(2)} out_j + w_{k0}^{(2)}$

The output for each neuron k at the output layer is given by:

- $out_k = \frac{e^{net_k}}{\sum_{j=1}^{K} e^{net_k}}$

The total error is given by:

- $E(\mathbf{w}) = \sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log out_k - \frac{\lambda}{2} ||\mathbf{w}||^2$

Using the above equations we can now calculate the derivatives of the chain rule (12):

$$\dashrightarrow \frac{\partial net_j}{\partial w_{ji}^{(1)}} = \frac{\partial \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}}{\partial w_{ji}^{(1)}} = x_i$$

For each of the three hidden layer activation functions we have:

1. $\frac{\partial out_j}{\partial net_j} = \frac{\partial log(1+e^{net_j})}{\partial net_j} = \frac{e^{net_j}}{1+e^{net_j}}$

2. $\frac{\partial out_j}{\partial net_j} = \frac{\partial \left(\frac{e^{net_j} - e^{-net_j}}{e^{net_j} + e^{-net_j}} \right)}{\partial net_j} = 1 - out_j^2$

3. $\frac{\partial out_j}{\partial net_j} = \frac{\partial cos(net_j)}{\partial net_j} = -sin(net_j)$

$$\dashrightarrow \frac{\partial net_k}{\partial out_j} = \frac{\partial \sum_{j=1}^{M} w_{kj}^{(2)} out_j + w_{k0}^{(2)}}{\partial out_j} = w_{kj}$$

Lastly, from the chain rule we know that $\frac{\partial E(\mathbf{w})}{\partial net_k} = \frac{\partial E(\mathbf{w})}{\partial out_k} * \frac{\partial out_k}{\partial net_k}$

but from (11) we have $\frac{\partial E(\mathbf{w})}{\partial net_k} = t_k - y_k$

$$\dashrightarrow \frac{\partial E(\mathbf{w})}{\partial out_k} * \frac{\partial out_k}{\partial net_k} = t_k - y_k$$

3 Implementation

We implemented a **SNN** in the **Python(3.6)** programming language in order to benefit from its great expressiveness and library support. To perform all the matrix and vector math, we used the widely used and highly optimized library **Numpy**, It offers great facilities for efficiently operating on matrix and vector data. Here we provide a high level overview of the implementation.

The source code is structured as follows

`Source/main.py`
`Source/snn/neuralnet.py`
`Source/snn/input.py`
`Source/snn/math.py`

3.1 Input

Data input is handled in **`input.py`**. The procedures are specifically implemented for efficiently loading the **`mnisttxt`** dataset as it was provided from the e-class platform. Since training data are given separately for each handwritten digit, a convenience routine was implemented that accepts a folder path where the 10 corresponding files are expected to be found. The same routine can be used for loading the test sample data.

The function signature is

```
def LoadFeaturesFromDirectory(Dirname, CountPerFile, Normalize)
```

Dirname specifies the directory path, **CountPerFile** specifies the number of features to load from each file (useful for debugging) and **Normalize** is a boolean value that specifies whether to normalize the data by dividing with the maximum value. In the context of the MNIST dataset, the max value is most likely 255.

The function, later queries all the files in the directory and calls the function

```
def LoadFeaturesFromFile(Filename, HotIndex, Count, Normalize):
```

that loads all Features and their Corresponding Labels. **HotIndex** is used to construct the one hot vector that will be used for labeling each of the train data in the file. **Count** is used as a convenience for specifying the exact number of features to read.

In the end, a tuple is returned containing the combined Features and Labels from all files.

3.2 Neural Network

We define the class **NeuralNet** to model a Neural Network. All functionality required to train and test the data is in this class.

3.2.1 NeuralNet

The class **NeuralNet** in **neuralnet.py** encapsulates all the functionality and state that characterizes a Neural Network. At a high level, the Neural Network consists of a number of **Layers** and the set of **hyper-parameters** used for training. For each forward pass, we start from the zero-th layer and perform the calculation until we reach the output layer where we initiate the backpropagation step.

The constructor of the class takes the form

```
def __init__(self, Structure, Activation=snn.math.sigmoid):
```

Structure is an integer array that holds the node count for each layer. In this particular implementation it is expected to be of length 3, holding the values for the input, hidden and output layer. The activation function is determined through the command line arguments. The passed function also dictates the derivative function that will be used.

The **train** function is the entry point for the neural network. It takes all the hyper-parameters that are used during training.

```
def train( self, TrainData, Epochs,
MiniBatchSize, LearningRate, LambdaNormalization, TestData = None ):
```

The implementation supports stochastic mini-batch gradient ascent and the size of each batch is provided in **MiniBatchSize**. The method loops over the **Epoch** count and slices the input data according to the batch size. These are then fed forward into the network for an initial prediction.

```
def feedforward( self, inputdata ):
```

The **feedforward** call takes a batch of data and passes it through the network returning the **softmax** result of the output layer. The returned values are later used the backpropagation step. With the result of the feed forward step, the Total Cost(log-likelihood) is also updated.

```
def backpropagate( self, Targets ):
```

The backpropagate function takes the expected targets and calculates the deltas according to the cross-etropy errors as covered in the previous section. The weights and bias deltas are held separately and for each layer. In the end, the function returns those deltas for all the features in the mini-batch.

These deltas are later scaled and added to the weights of the corresponding layer so that the neural network is trained for the next rounds. When updating the each Layer's weights, we make sure to also scale by the learning rate and account for the lambda normalization.

3.2.2 Math

The **math** submodule contains all the functions that we might need. It provides implementation for all the supported activation functions and for their derivatives. **Numpy** greatly helps in this regard as it seemlessly allows for applying all math operations on vector and scalar values alike.

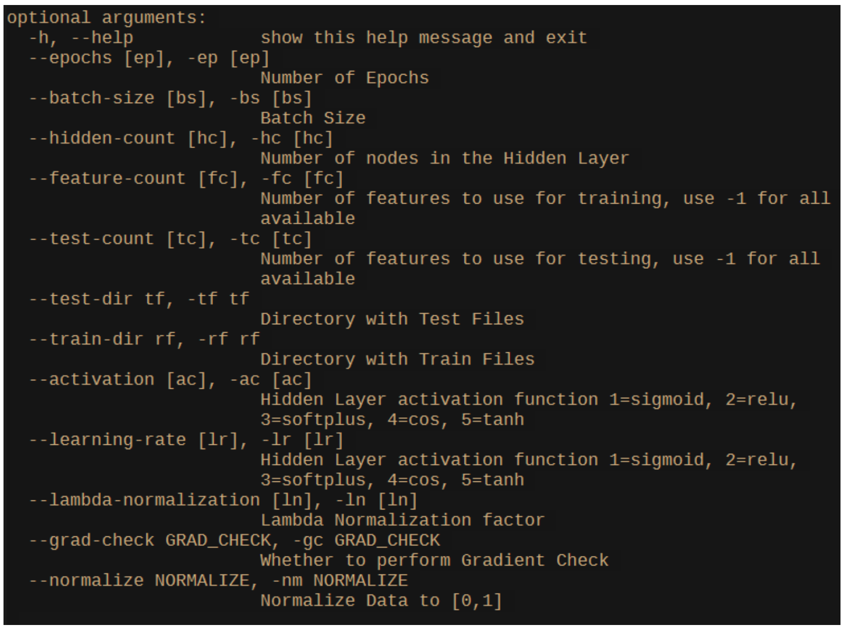
This file also contains the code for executing the gradient check if the corresponding flag is passed to the command line.

In general, files are populated with comments describing all functionality and decisions that were made.

4 Experiments

We present some of the training results of SNN. In all of the cases, we use the entire mnist dataset (i.e. >50000 feature vectors and 10000 test data):

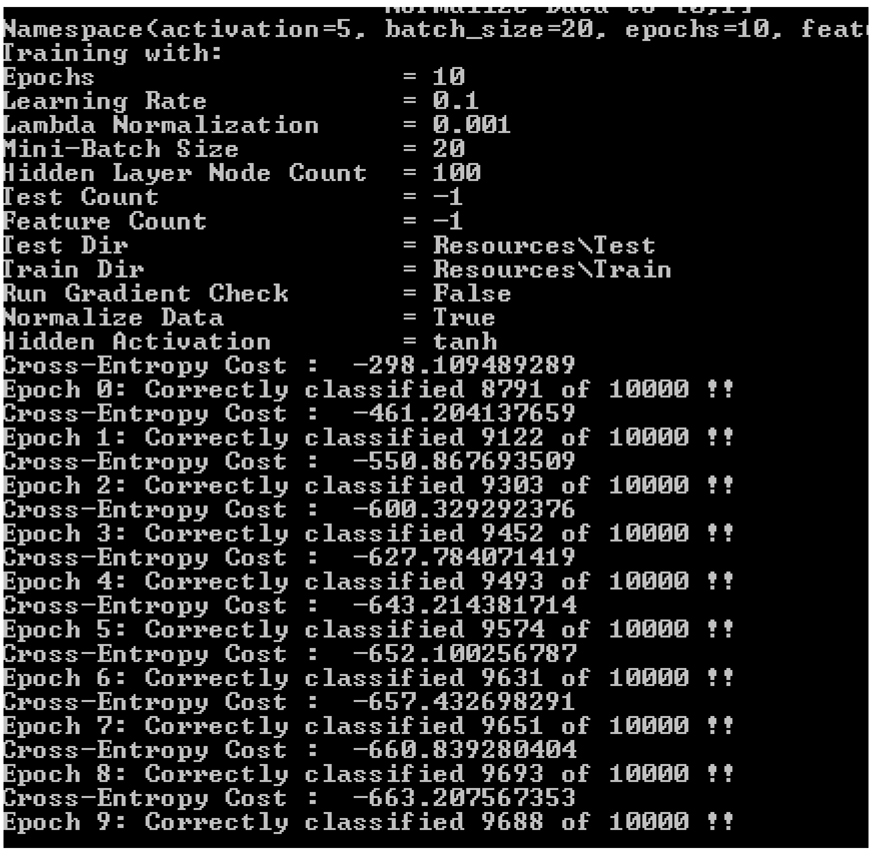
In Figure 3 we see all the supported command line arguments.



```
optional arguments:
  -h, --help            show this help message and exit
  --epochs [ep], -ep [ep]
                        Number of Epochs
  --batch-size [bs], -bs [bs]
                        Batch Size
  --hidden-count [hc], -hc [hc]
                        Number of nodes in the Hidden Layer
  --feature-count [fc], -fc [fc]
                        Number of features to use for training, use -1 for all
                        available
  --test-count [tc], -tc [tc]
                        Number of features to use for testing, use -1 for all
                        available
  --test-dir tf, -tf tf
                        Directory with Test Files
  --train-dir rf, -rf rf
                        Directory with Train Files
  --activation [ac], -ac [ac]
                        Hidden Layer activation function 1=sigmoid, 2=relu,
                        3=softplus, 4=cos, 5=tanh
  --learning-rate [lr], -lr [lr]
                        Hidden Layer activation function 1=sigmoid, 2=relu,
                        3=softplus, 4=cos, 5=tanh
  --lambda-normalization [ln], -ln [ln]
                        Lambda Normalization factor
  --grad-check GRAD_CHECK, -gc GRAD_CHECK
                        Whether to perform Gradient Check
  --normalize NORMALIZE, -nm NORMALIZE
                        Normalize Data to [0,1]
```

Figure 3: Available options

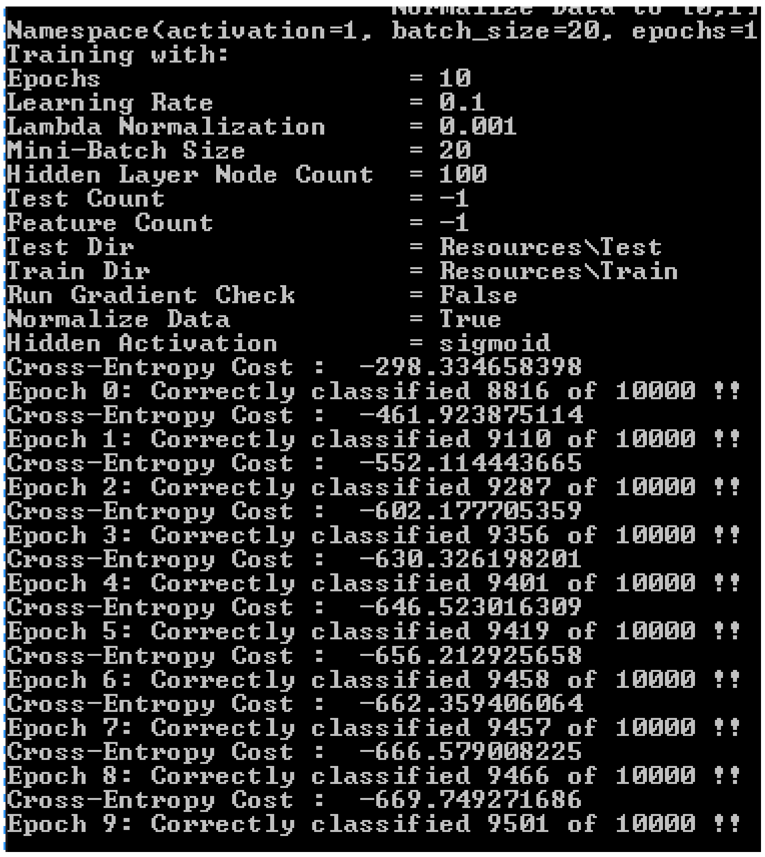
Figure 4 shows a train session using the corresponding hyper-parameters. As we can see the **tanh** function achieves very low error rate.



```
Namespace(activation=5, batch_size=20, epochs=10, feat
Training with:
Epochs                     = 10
Learning Rate              = 0.1
Lambda Normalization       = 0.001
Mini-Batch Size            = 20
Hidden Layer Node Count    = 100
Test Count                 = -1
Feature Count              = -1
Test Dir                   = Resources\Test
Train Dir                  = Resources\Train
Run Gradient Check         = False
Normalize Data             = True
Hidden Activation          = tanh
Cross-Entropy Cost :  -298.109489289
Epoch 0: Correctly classified 8791 of 10000 !!
Cross-Entropy Cost :  -461.204137659
Epoch 1: Correctly classified 9122 of 10000 !!
Cross-Entropy Cost :  -550.867693509
Epoch 2: Correctly classified 9303 of 10000 !!
Cross-Entropy Cost :  -600.329292376
Epoch 3: Correctly classified 9452 of 10000 !!
Cross-Entropy Cost :  -627.784071419
Epoch 4: Correctly classified 9493 of 10000 !!
Cross-Entropy Cost :  -643.214381714
Epoch 5: Correctly classified 9574 of 10000 !!
Cross-Entropy Cost :  -652.100256787
Epoch 6: Correctly classified 9631 of 10000 !!
Cross-Entropy Cost :  -657.432698291
Epoch 7: Correctly classified 9651 of 10000 !!
Cross-Entropy Cost :  -660.839280404
Epoch 8: Correctly classified 9693 of 10000 !!
Cross-Entropy Cost :  -663.207567353
Epoch 9: Correctly classified 9688 of 10000 !!
```

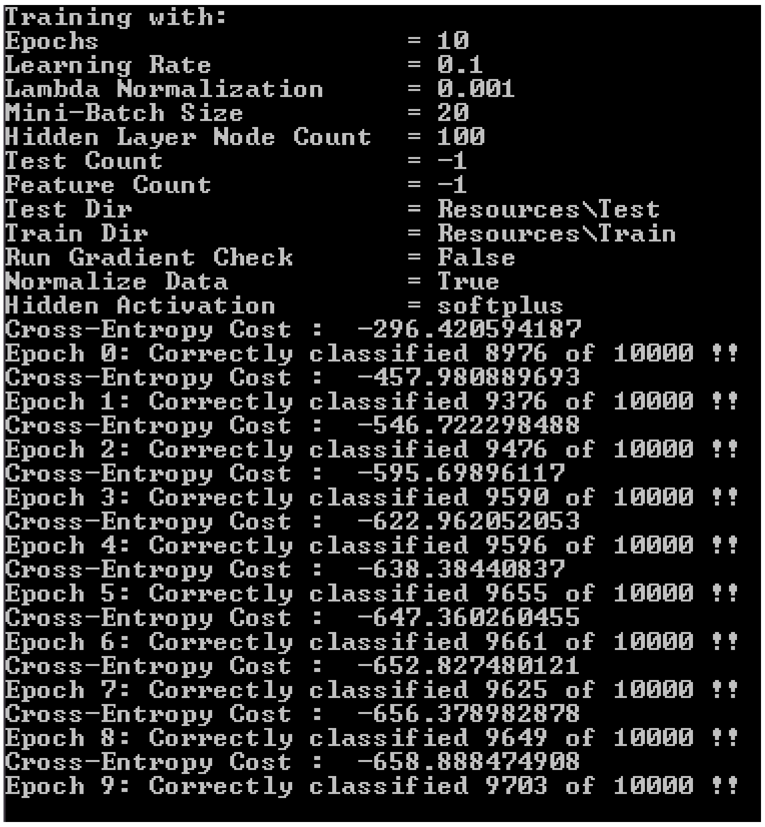
Figure 4: tanh activation function

Similarly, in Figures 5 and 6 we show a train session using the **sigmoid** and **softplus** activation functions respectively, which achieve great performance.



```
Namespace(activation=1, batch_size=20, epochs=1
Training with:
Epochs                    = 10
Learning Rate             = 0.1
Lambda Normalization      = 0.001
Mini-Batch Size           = 20
Hidden Layer Node Count   = 100
Test Count                = -1
Feature Count             = -1
Test Dir                  = Resources\Test
Train Dir                 = Resources\Train
Run Gradient Check        = False
Normalize Data            = True
Hidden Activation         = sigmoid
Cross-Entropy Cost :  -298.334658398
Epoch 0: Correctly classified 8816 of 10000 !!
Cross-Entropy Cost :  -461.923875114
Epoch 1: Correctly classified 9110 of 10000 !!
Cross-Entropy Cost :  -552.114443665
Epoch 2: Correctly classified 9287 of 10000 !!
Cross-Entropy Cost :  -602.177705359
Epoch 3: Correctly classified 9356 of 10000 !!
Cross-Entropy Cost :  -630.326198201
Epoch 4: Correctly classified 9401 of 10000 !!
Cross-Entropy Cost :  -646.523016309
Epoch 5: Correctly classified 9419 of 10000 !!
Cross-Entropy Cost :  -656.212925658
Epoch 6: Correctly classified 9458 of 10000 !!
Cross-Entropy Cost :  -662.359406064
Epoch 7: Correctly classified 9457 of 10000 !!
Cross-Entropy Cost :  -666.579008225
Epoch 8: Correctly classified 9466 of 10000 !!
Cross-Entropy Cost :  -669.749271686
Epoch 9: Correctly classified 9501 of 10000 !!
```

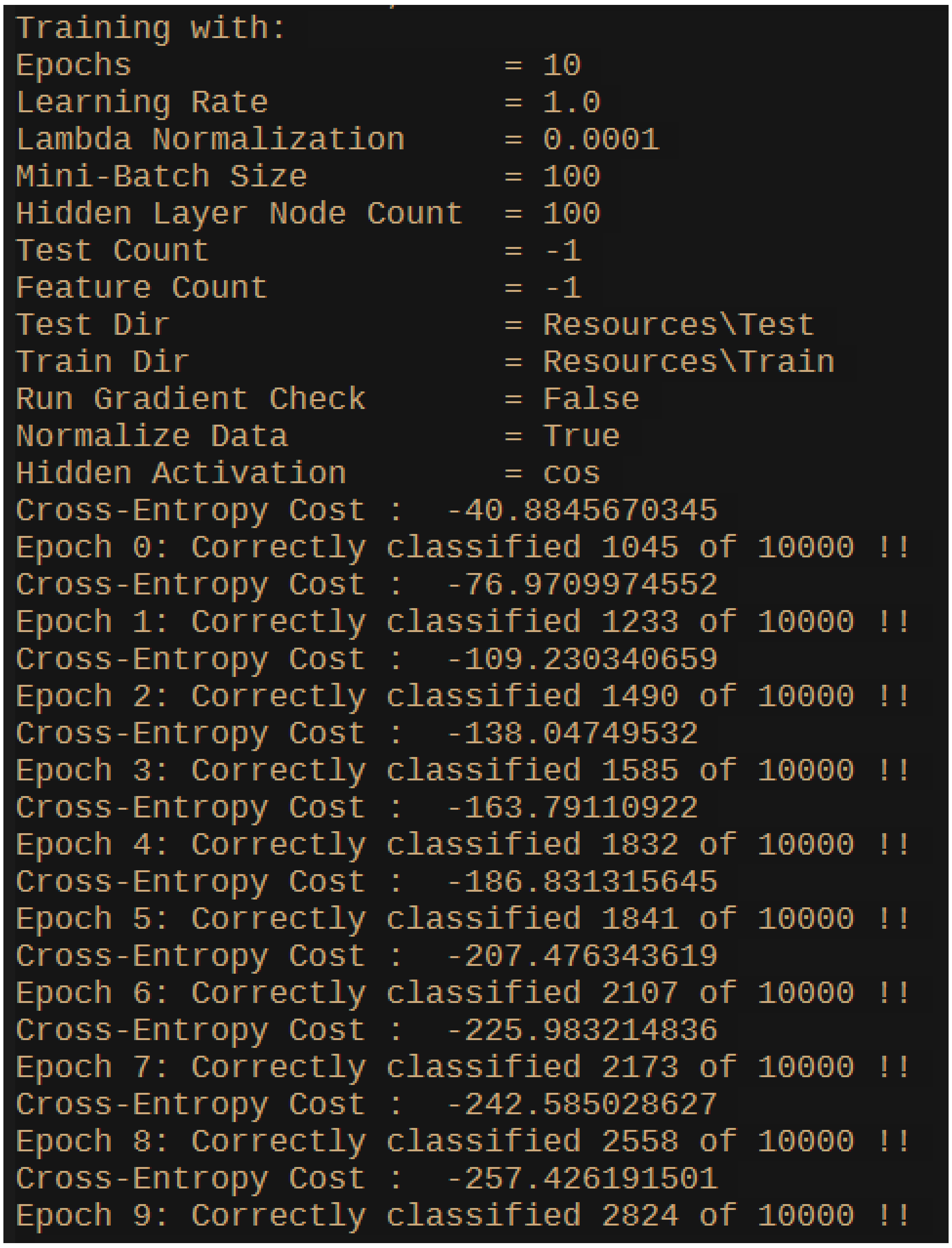
Figure 5: sig activation function



```
Training with:
Epochs                    = 10
Learning Rate             = 0.1
Lambda Normalization      = 0.001
Mini-Batch Size           = 20
Hidden Layer Node Count   = 100
Test Count                = -1
Feature Count             = -1
Test Dir                  = Resources\Test
Train Dir                 = Resources\Train
Run Gradient Check        = False
Normalize Data            = True
Hidden Activation         = softplus
Cross-Entropy Cost :  -296.420594187
Epoch 0: Correctly classified 8976 of 10000 !!
Cross-Entropy Cost :  -457.980889693
Epoch 1: Correctly classified 9376 of 10000 !!
Cross-Entropy Cost :  -546.722298488
Epoch 2: Correctly classified 9476 of 10000 !!
Cross-Entropy Cost :  -595.69896117
Epoch 3: Correctly classified 9590 of 10000 !!
Cross-Entropy Cost :  -622.962052053
Epoch 4: Correctly classified 9596 of 10000 !!
Cross-Entropy Cost :  -638.38440837
Epoch 5: Correctly classified 9655 of 10000 !!
Cross-Entropy Cost :  -647.360260455
Epoch 6: Correctly classified 9661 of 10000 !!
Cross-Entropy Cost :  -652.827480121
Epoch 7: Correctly classified 9625 of 10000 !!
Cross-Entropy Cost :  -656.378982878
Epoch 8: Correctly classified 9649 of 10000 !!
Cross-Entropy Cost :  -658.888474908
Epoch 9: Correctly classified 9703 of 10000 !!
```

Figure 6: softplus activation function

However, the same train session using the **cos** activation function shows very bad results as we can see in Figure 7.



```
Training with:
Epochs                     = 10
Learning Rate              = 1.0
Lambda Normalization       = 0.0001
Mini-Batch Size            = 100
Hidden Layer Node Count    = 100
Test Count                 = -1
Feature Count              = -1
Test Dir                   = Resources\Test
Train Dir                  = Resources\Train
Run Gradient Check         = False
Normalize Data             = True
Hidden Activation          = cos
Cross-Entropy Cost :  -40.8845670345
Epoch 0: Correctly classified 1045 of 10000 !!
Cross-Entropy Cost :  -76.9709974552
Epoch 1: Correctly classified 1233 of 10000 !!
Cross-Entropy Cost :  -109.230340659
Epoch 2: Correctly classified 1490 of 10000 !!
Cross-Entropy Cost :  -138.04749532
Epoch 3: Correctly classified 1585 of 10000 !!
Cross-Entropy Cost :  -163.79110922
Epoch 4: Correctly classified 1832 of 10000 !!
Cross-Entropy Cost :  -186.831315645
Epoch 5: Correctly classified 1841 of 10000 !!
Cross-Entropy Cost :  -207.476343619
Epoch 6: Correctly classified 2107 of 10000 !!
Cross-Entropy Cost :  -225.983214836
Epoch 7: Correctly classified 2173 of 10000 !!
Cross-Entropy Cost :  -242.585028627
Epoch 8: Correctly classified 2558 of 10000 !!
Cross-Entropy Cost :  -257.426191501
Epoch 9: Correctly classified 2824 of 10000 !!
```

Figure 7: cos activation function

References

[ANNa] Intro to neural networks.
`https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/`.
Accessed: 11-06-2017.

[ANNb] Neural networks.
`http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html`. Accessed:
11-06-2017.

[Bis07] C Bishop. Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn. *Springer, New York*, 2007.