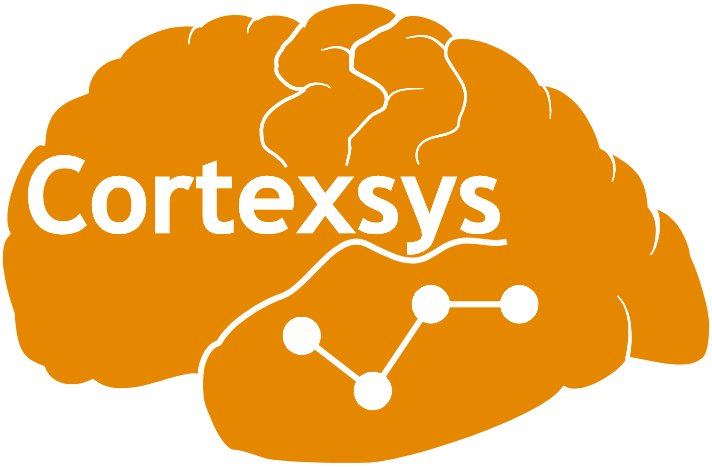
Cortexsys 3.0 User Guide

February, 2016



Sandia National Laboratories

U.S. DEPARTMENT OF ENERGY

Authorship and Acknowledgements

The Cortexsys 1.0, 2.0 and 3.0 code and user guide was written by Jonathan A. Cox (jacox@sandia.gov, joncox@alum.mit.edu) at Sandia National Laboratories under the support of the Laboratory Directed Research and Development (LDRD) program. The authors wish to acknowledge helpful discussions and feedback from Timothy J. Draelos.

All algorithms in the code are implementations of those which have been openly published in the academic literature. Please refer to the references at the end of this document for further information.

License

Contents

Introduction

Cortexsys is a deep learning toolbox for Matlab and GNU Octave 4.0 intended for researchers and algorithm developers who would like to rapidly implement and analyze new algorithms with the Matlab or Octave environment. Cortexsys strives to accomplish too goals which are often in conflict: (1) Easy to use and learn; (2) Flexible and adaptable for research, education and prototyping. If we have achieved this goal, a new user should be able to instantly run and understand the examples, and eventually dig into and understand the underlying "guts" without excessive confusion.

Benefits of Cortexsys

- Matrix and mathematical oriented language and development environment
    - For example, multiplying a matrix and a vector and adding a constant is accomplished simply with the commands: `A*b+c`
    - Completely interpreted language provides easy and fast debugging
    - Interoperability with the wealth of existing toolboxes and code
- Easy to combine various types of neural network layers together
    - For example, a convolutional net can feed a recurrent net that outputs to a fully connected feed-forward net.
- Simple support for GPU accelerator cards and cluster computing with the Matlab Parallel Computing Toolbox.
- Explicit implementation of the training and backpropagation algorithms that does not use automatic differentiation. This is useful for optimization, educational purposes and porting code to compiled software or dedicated hardware.

At present, Cortexsys implements both standard, feed forward deep neural nets ("deep learning"), including convolutional nets, and recurrent neural nets. Furthermore, all types can be combined in arbitrary ways[1].

Summary of Features

- Layer types
    - Fully connected feed forward
    - Convolutional
    - Average pooling
    - Recurrent
    - Long-short term memory (LSTM)
- Activation functions
    - Sigmoid
    - Hyperbolic tangent
    - Softplus
    - Softmax
    - Linear
    - Rectified linear

[1] Combining convolutional nets and recurrent nets will be added back into version 3.1.

- Leaky rectified linear
- Optimization algorithms
  - Mini-batch gradient descent with momentum
  - ADADELTA gradient descent
  - Mini-batch conjugate gradient with line-search
- Regularization
  - L_2 norm of weights
  - Dropout
  - De-noising of input
  - Max-norm
  - Sparsity constraints
  - Tied weights (for auto-encoders)
- Learning types
  - Supervised classifiers
  - Unsupervised auto-encoders
  - Sequence prediction (recurrent nets)
  - Unsupervised layer-by-layer pre-training
  - Input maximization for classifiers
    - Learn an input that maximizes the activation for a particular unit activation, such as the output units. This allows us to generate "optical illusions" or to visualize what various features may represent.
- Examples
  - Deep auto-encoder with pre-training
  - Deep classifier with pre-training
  - Convolutional net for image classification
  - Recurrent network for sequence prediction
    - Generate Shakespeare text via random sampling
  - Neural net "optical illusion" generator
    - Generate inputs that an MNIST classifier believes are digits with >99.9% confidence

Feature To-do List

In addition to the features above, the following features may be implemented at some point in the future, and will be a useful addition to the code base.

- Layers
  - Max pooling layers for convolutional nets
- Optimization algorithms
  - Stochastic Hessian Free Optimization for deep and recurrent nets
- Regularization
  - Mini-batch normalization of layer outputs
- Examples
  - Image caption generator with recurrent net

Initialization and Startup

Before launching Cortexsys, it is necessary to compile the CUDA routines if convolutional neural nets are being used and GPU acceleration is desired. The MMX routines must be compiled if using recurrent networks with any network type, unless GPU acceleration is being used. These routines accelerate Matlab's built in functions for specific operations where built-in routines were either unavailable or did not have the desired performance.

To compile these routines, you must setup your MEX compiler (see Matlab documentation[2]). To compile the CUDA routines for convolutional nets, you must also have the NVIDIA CUDA Toolkit[3] installed.

1. Enter the `nn_core/mmx` directory and run the `build_mmx.m` script.
2. Enter the `nn_core/cuda` directory and run the `cudaBuild_and_Test.m` script. This will compile the cuda routines and perform some numerical tests for accuracy and performance.

Before calling any Cortexsys routines, you must also add the necessary directories to your Matlab path. The examples use the following code to accomplish this. You must adjust the root part of the path '../../' to reflect the location of the Cortexsys directory (e.g. '~/user/Cortexsys/').

```
addpath('../../nn_gui');
addpath('../../nn_core');
addpath('../../nn_core/cuda');
addpath('../../nn_core/mmx');
addpath('../../nn_core/Optimizers');
addpath('../../nn_core/Activations');
addpath('../../nn_core/Wrappers');
addpath('../../nn_core/ConvNet');
```

Next, you must initialize Cortexsys and setup some basic parameters that are stored in the `definitions` object. This object must be passed to many of the Cortexsys objects and routines. Call the definitions constructor:

```
defs = definitions(PRECISION, useGPU, whichGPU, plotOn);
```

Here, PRECISION is a string that may be either 'double' or 'single'. Also, `useGPU` and `plotOn` are booleans (set to true or false) that control whether plotting is enabled and if a GPU should be used. If a GPU is used, `whichGPU` is the number of the GPU card to use.

[2] http://www.mathworks.com/support/compilers/R2015b/index.html
[3] https://developer.nvidia.com/cuda-toolkit

Input and Target (Output) Data

Input and output data (as well as layer activations and some other types of data) is stored in a `varObj` object. All objects in Cortexsys are "handle objects", which are simply pointers to objects. Therefore, if you have a `varObj` called `A`, and you set `B = A`, both `B` and `A` now point to the same object with the same data. The internal data is stored in the public `v` member, which is accessed like `A.v`. When defining a varObj, you can optionally specify the type of data, such as `INPUT_TYPE` or `OUTPUT_TYPE`. If the data is of one of these types, you can extract a mini-batch from the data and automatically load it into the GPU with the `getmb()` method: `A.getmb()`. Note, the data stored in the `varObj` may be a sparse matrix in memory, from which a mini-batch is extracted and loaded into the GPU.

Cortexsys is also capable of handling both single precision and double precision floating point data. This can be useful for some inexpensive GPU cards that cannot perform double precision calculations quickly. For convenience, the `precision(A, defs)` function will convert a raw matrix or tensor to a particular type depending on what was set during initialization in the `definitions` object.

Layer Structure and Network Architecture

Neural networks in Cortexsys are divided up into layers. Each layer may be of a different type, such as "fully connected" or "LSTM". This is defined in a layers structure that determines the activation function, the size of the layer and the type of the layer.

A typical feed-forward, fully connected MNIST classifier network can be as shown below. This network has LReLU (leaky rectified linear) activation functions and has a softmax output layer with a cross-entropy cost function. The size of each layer, set in layers.sz, is a three element array. Except for convolutional layers, only the first element can be other than 1. For instance, [768 1 1] creates a layer with 768 LReLU units. A convolutional layer of size [12 5 5] would have 12 feature maps with kernel sizes of 5x5 pixels each.

```
layers.af{1} = [];
layers.sz{1} = [input_size 1 1];
layers.typ{1} = defs.TYPES.INPUT;

layers.af{end+1} = LReLU(defs, defs.COSTS.SQUARED_ERROR);
layers.sz{end+1} = [768 1 1];
layers.typ{end+1} = defs.TYPES.FULLY_CONNECTED;

layers.af{end+1} = LReLU(defs, defs.COSTS.SQUARED_ERROR);
layers.sz{end+1} = [256 1 1];
layers.typ{end+1} = defs.TYPES.FULLY_CONNECTED;

layers.af{end+1} = softmax(defs, defs.COSTS.CROSS_ENTROPY);
layers.sz{end+1} = [output_size 1 1];
layers.typ{end+1} = defs.TYPES.FULLY_CONNECTED;
```

In contrast to some approaches, the input data is represented as layer 1, and is effectively just another layer in the network.

It is possible for layers other than the output layer to have a cost function. This is necessary when layer-by-layer pre-training is being used, so that each layer can be trained as an unsupervised auto-encoder.

Visualizing the Network

The connectivity, type and size of the layers can be visualized with the `nnShow` method located in the nn_gui folder. Simply pass `nnShow` with a figure number, the layers structure and the definitions object.

```
nnShow(123, layers, defs);
```

For the recurrent net example, the following plot is generated:

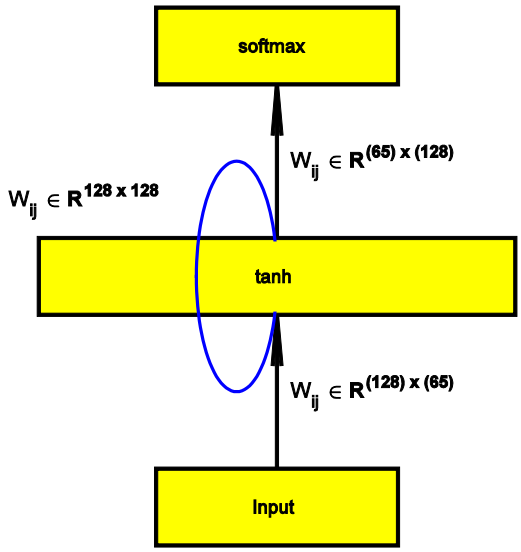


Figure 1: Diagram of the recurrent net example. The dimensionality of the weights is shown, along with a blue ellipse indicating that the middle layer with hyperbolic tangent activation function is recurrent. The width of the layers represents the number of units in the layer.

Training Parameters

A variety of settings exist that control how the network is trained. Examples include the learning rate, dropout regularization, sparsity penalties, tied weights, etc. These parameters must be defined in a structure and then passed to the `nnLayers` object when the network object is created (to the constructor). In general, the values of these hyper-parameters are not known in advance, and it is best to play with different values to get a feeling for what works best.

```
% Training parameters
params.maxIter = 1000; % How many iterations to train for
params.miniBatchSize = 128; % set size of mini-batches

% Regularization parameters
params.maxnorm = 0; % enable max norm regularization if ~= 0
params.lambda = 1; % enable L2 regularization if ~= 0
params.alphaTau = 0.25*params.maxIter; % Learning rate decay
params.denoise = 0.25; % enable input denoising if ~= 0
params.dropout = 0.6; % enable dropout regularization if ~= 0
params.tieWeights = false; % enable tied weights for autoencoder?
params.beta_s = 0; % Strength of sparsity penalty; set to 0 to disable
params.rho_s0 = 0; % Target hidden unit activation for sparsity penalty

% Learning rate parameters
params.momentum = 0.9; % Momentum for stochastic gradient descent
params.alpha = 0.01; % Learning rate for SGD
params.rho = 0.95; % AdaDelta hyperparameter
params.eps = 1e-6; % AdaDelta hyperparameter

% Conjugate gradient parameters
params.cg.N = 10; % Max CG iterations before reset
params.cg.sigma0 = 0.01; % CG Secant line search parameter
params.cg.jmax = 10; % Maximum CG Secant iterations
params.cg.eps = 1e-4; % Update threshold for CG
params.cg.mbIters = 10; % How many CG iterations per minibatch?
```

If the data is time series data, where each time step is stored in a different cell in a cell array, `getmb()` extracts the mini-batch and converts it into a 3D tensor, where the third dimension is time.

Creating a Network Object

Once the initialization object, layer topology, input and output data and training parameters are defined, the neural network object can be created. This object ties together all of these aspects and represents everything about the neural net. It is also used to store some persistent data during training, such as layer activations and dropout masks.

The `nnLayers` object stores the layer structure, as well as references to the input and output data, and all of the parameters associated with training. It also has the job of initializing the layer weights. The weights and biases are also stored in this object.

The nnLayers object expects one structure containing the layers, including another containing parameters for training. It can optionally accept the input and output data objects and initial weights and biases. For example, the object can be created and the weights and biases initialized with the following code:

```
nn = nnLayers(params, layers, X, Y, {}, {}, defs);
nn.initWeightsBiases();
```

Or, to use your own initial weights and biases, you can pass `nnLayers` `W` and `b` at creation:

```
nn = nnLayers(params, layers, X, Y, W, b, defs);
```

Keep in mind, the input and target data, X and Y, could be the same `varObj` to save memory—as in the case of an auto-encoder.

Training a Neural Network

Once you have defined your nnLayers object, you can train this network using one of several optimization algorithms. At this time, stochastic mini-batch gradient descent, ADADELTA gradient descent, and stochastic mini-batch conjugate gradient with line search is provided.

First, create a function handle to the function that the optimization routine will use to compute the gradients for your network. This function uses the backpropagation algorithm to compute the gradients of the weights and biases with respect to the cost function defined in the layers structure.

```
costFunc = @(nn,r,newRandGen) nnCostFunctionCNN(nn,r,newRandGen);
```

The function handle takes three arguments: the nnLayers object, the mini-batch to use (indicies of data to extract from the data set), and whether or not randomly generated elements of the network should be re-drawn (e.g. dropout or denoising). The second and third parameters are useful for optimization routines that must control the stochastic nature of the network during training, such as conjugate gradient.

Next, pass the training function to the optimization routine:

```
gradientDescentAdaDelta(costFunc, nn, defs, Xts, Yts, yts, y, 'Training Entire
Network');
```

The routine expects the cost function handle, the nnLayers object and the definitions object. Optionally, you can also provide cross-validation (test sets) for checking the accuracy during training. A figure title is provided so that different stages of training can be identified, when plotting the training progress.

Evaluating a Trained Network

Once a network is trained, it can be used to predict a sequence or otherwise generate an output, such as a class prediction.

Non-recurrent Nets

To compute the output of a network on a set of input data, load the appropriate data into the first layer in the `nnLayers` object and pass it to the `feedforward` function.

```
nn.A{1} = Xts;
nn.Y = Yts;
m = size(Yts.v, 2);
Aout = feedForward(nn, m);
[pred, probs]= predictDeep(A, false, true));
acc = mean(double(pred == yts)) * 100;
```

`feedforward` takes the number of examples in the data set, `m`, and returns the output of the final layer. This output can be passed to the `predictDeep` function for evaluating the accuracy of a classifier. The predictDeep function simply makes the maximum likelihood prediction.

Recurrent Nets

Evaluating the prediction of a recurrent network is a bit more complicated. Please consult the end of the LSTM_RNN example Shakespeare generator for more details. However, the `feedForwardLSTM` method for a recurrent net also takes as an argument the particular time step to operate on. If you place `feedForwardLSTM` in a loop, and increment the time step *t*, it will advance the network one time step at a time. For this example, the network expects to receive as input the output from the previous time step. Thus, in between evaluating time steps, we set the input at `A{1}` to be the previous time step's output prediction.

```
for t= 2:T_samp
    nn.A{1}.v(:,:,t) = Aout(:,:,t-1); % NOT ACTUALLY DONE THIS WAY
    % Step the RNN forward
    Aout = feedforwardLSTM(nn, 1, t, false, true);
end
```

Additionally, there are some details with respect to how a recurrent net's output must be evaluated, in this case. Instead of simply making the maximum likelihood prediction, as above, it is necessary to draw (or sample) from the probability distribution provided by the network prediction. Otherwise, if this process is deterministic, this network will make deterministic and uninteresting predictions.

The Matlab `randsample` function, from the Statistics and Machine Learning Toolbox, is used to draw a random sample from the output distribution. This choice represents a choice of a character. If this toolbox is not available, Octave does have an open source implementation.

Recurrent Networks

Recurrent networks include a time-domain aspect that can operate on a sequence of data. For example, the input could be a video, where each time step is a different image. Alternatively, each time step could be a word or character from a text, such as Shakespeare's works.

When working with recurrent nets, an additional parameter that sets the length of the time sequence must be defined.

```
params.T = 50; % Length of sequence is 50 time steps
```

Keep in mind, time series data is actually represented as a sequence of length T+2 (see Data Representations).

Another parameter, T_{OS}, controls whether the initial output of the net is factored into the training and the computed cost. For example, it may not be helpful to expect a Shakespeare sequence generator to accurately predict an entire sentence of text based only on the first character in the sentence. In this case, if T_{OS} is not zero, the first *n* time steps will not count toward the network training. This gives you the ability to "prime" the net with a sequence before optimizing the predicted sequence.

```
params.Tos = 5; % Length of sequence to ignore when training
```

Data Representations

Generally speaking, there are three types of input, output and activation layer data. Again, this data is commonly stored in a `varObj` object.

1. 2D, *n* x *m* matrix for non-recurrent, non-convolutional data
2. 4D k_1 x k_2 x *n* x *m* tensor for non-recurrent, convolutional data
3. 1D Cell array (with *t* elements) of 2D *n* x *m* matrices for recurrent, time-domain data
    a. This is converted internally to an *n* x *m* x *t* tensor by the `varObj::getmb()` method.
4. 1D Cell array (with *t* elements) of 4D k_1 x k_2 x *n* x *m* tensors for recurrent, time-domain data that is convolutional/image/feature map data.

In the above, ***m*** is the index of the training example (m = number of training examples or mini-batch size). ***n*** is the dimensionality of the data or number of feature maps (convolutional net). ***t*** is the time step of the sequence. For convolutional/image data, $\boldsymbol{k_1}$ and $\boldsymbol{k_2}$ are the dimensions of the image or convolutional kernel.

Non-time Series Data

For example, the MNIST set would be a 2D matrix of 784 x 60000. Represented to a convolutional net, MNIST data would be 28 x 28 x 1 x 60000 ($784 = 28^2$). The output of a convolutional layer with 5 feature maps and 5x5 kernels could be 23 x 23 x 5 x 60000, if operating directly on the MNIST data.

Time Series Data

For recurrent (time series) data, each time step is stored as an element in a 1D cell array. Each element in the cell array is an identically sized matrix or tensor (just like as above) representing the data at that time step.

Keep in mind, time series data is actually represented as a sequence of length T+2, where *t* = 1 is the initial conditions (all zeros) and *t* = T+2 is the ending boundary conditions (all zeros). *t* = 2 is, in fact, the first time step in the series. This is necessary to simplify implementation. Since Matlab doesn't allow zero indexing of arrays, *t* = 0 is not the initial conditions, and so *t* = 2 must be the first real time step in the data.

Sparse Data

The input and output data can be stored in a sparse matrix. This is useful when a "one hot" or "n vs. k" representation is used. For instance, we can represent the MNIST target data as one of 10 possible values (one for each digit), where a 10 dimension vector with a "1" at the proper place represents the correct digit. Instead of having a large matrix with many zeros, a sparse representation can be used. The `varObj::getmb()` method automatically converts this to a full matrix and loads into the GPU, if enabled.

Network Parameter (Weight and Bias) Representations

Non-recurrent, Non-convolutional

The weight and bias parameters define the trained network. For a non-recurrent, non-convolutional network, the bias, `b`, is a column vector with dimensionality equal to the number of units in the layer. The weights, `W`, is a matrix with dimensionality n_2 x n_1, where n_1 is the number of units on the input layer, and n_2 is the number of the connecting layer above. Thus, `b` has dimensionality n_1.

Recurrent

For recurrent networks, the weights at every time step are identical. Therefore, they have the same shape as for the non-recurrent case.

Convolutional

For convolutional layers, the weights are 4D tensor of k_1 x k_2 x n_1 x n_2, where:

k_1 x k_2: kernel dimensions in x and y (convolutional filter)

n_1: number of feature maps in lower layer

n_2: number of feature maps in upper layer

It is helpful to think of convolutional weights and feature maps as the 2D analog of weights in a typical, fully connected net. So for a fully connected net, you can imagine that `W` is actually a *1* x *1* x n_2 x n_1 tensor, where the first two dimensions are singleton, and give simply a scalar weight value. Of course, for a convolutional net, this 2D "weight" is scanned across the input image or map.

Convolutional units also have biases, which are scalar values added to each feature map. Thus, if a layer has *n* feature maps, it will also have an *n* dimensional bias vector.

References

1. LeCun, Yann, and Yoshua Bengio. "Convolutional networks for images, speech, and time series." *The handbook of brain theory and neural networks* 3361.10 (1995): 1995.
2. Maas, Andrew L., Awni Y. Hannun, and Andrew Y. Ng. "Rectifier nonlinearities improve neural network acoustic models." *Proc. ICML*. Vol. 30. 2013.
3. Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958.
4. Vincent, Pascal, et al. "Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion." *The Journal of Machine Learning Research* 11 (2010): 3371-3408.
5. Sutskever, Ilya, et al. "On the importance of initialization and momentum in deep learning." *Proceedings of the 30th international conference on machine learning (ICML-13)*. 2013.
6. Hochreiter, Sepp, and Jürgen Schmidhuber. "Long short-term memory." *Neural computation* 9.8 (1997): 1735-1780.
7. Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313.5786 (2006): 504-507.
8. Ng, Andrew. "Sparse autoencoder." *CS294A Lecture notes* 72 (2011): 1-19.
9. Shewchuk, Jonathan Richard. "An introduction to the conjugate gradient method without the agonizing pain." (1994).
10. Karpathy, Andrej. "The unreasonable effectiveness of recurrent neural networks." (2015).
11. Zeiler, Matthew D. "ADADELTA: an adaptive learning rate method." *arXiv preprint arXiv:1212.5701* (2012).
12. Graves, Alex. *Supervised sequence labelling*. Springer Berlin Heidelberg, 2012.
13. Nguyen A, Yosinski J, Clune J. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In Computer Vision and Pattern Recognition (CVPR '15), IEEE, 2015.
14. Christopher, M. Bishop. "Pattern recognition and machine learning." Company New York 16.4 (2006): 049901.
15. Hopfield, John J. "Neural networks and physical systems with emergent collective computational abilities." Proceedings of the national academy of sciences 79.8 (1982): 2554-2558.
16. Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." Cognitive modeling 5.3 (1988): 1.
17. McCulloch, Warren S., and Walter Pitts. "A logical calculus of the ideas immanent in nervous activity." The bulletin of mathematical biophysics 5.4 (1943): 115-133.