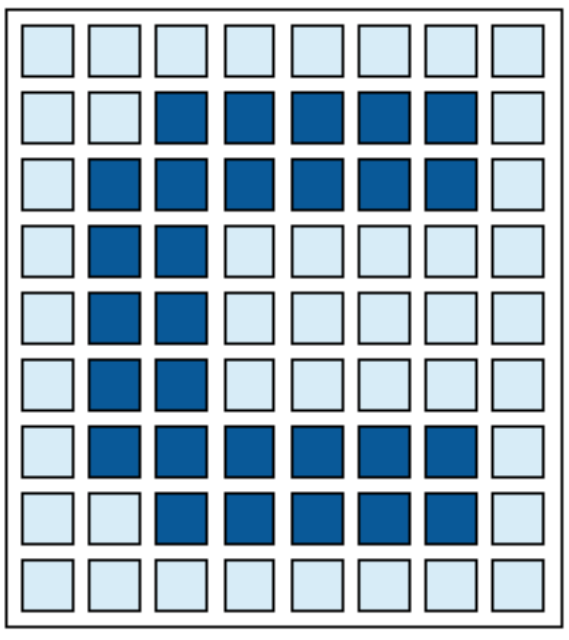
Deep Neural Network for MNIST Handwriting Recognition

I finally found some time to enhance my neural network to support deep learning. The network now masters a variable number of layers and is capable of running convolutional layers. The architecture is generic, light weight (very small memory footprint) and super fast. :-)



In a previous blog post I wrote about a simple 3-Layer neural network for MNIST handwriting recognition that I built. Its architecture – a 3-layer structure with exactly 1 hidden layer – was fix. And it only supported normal fully connected layers.

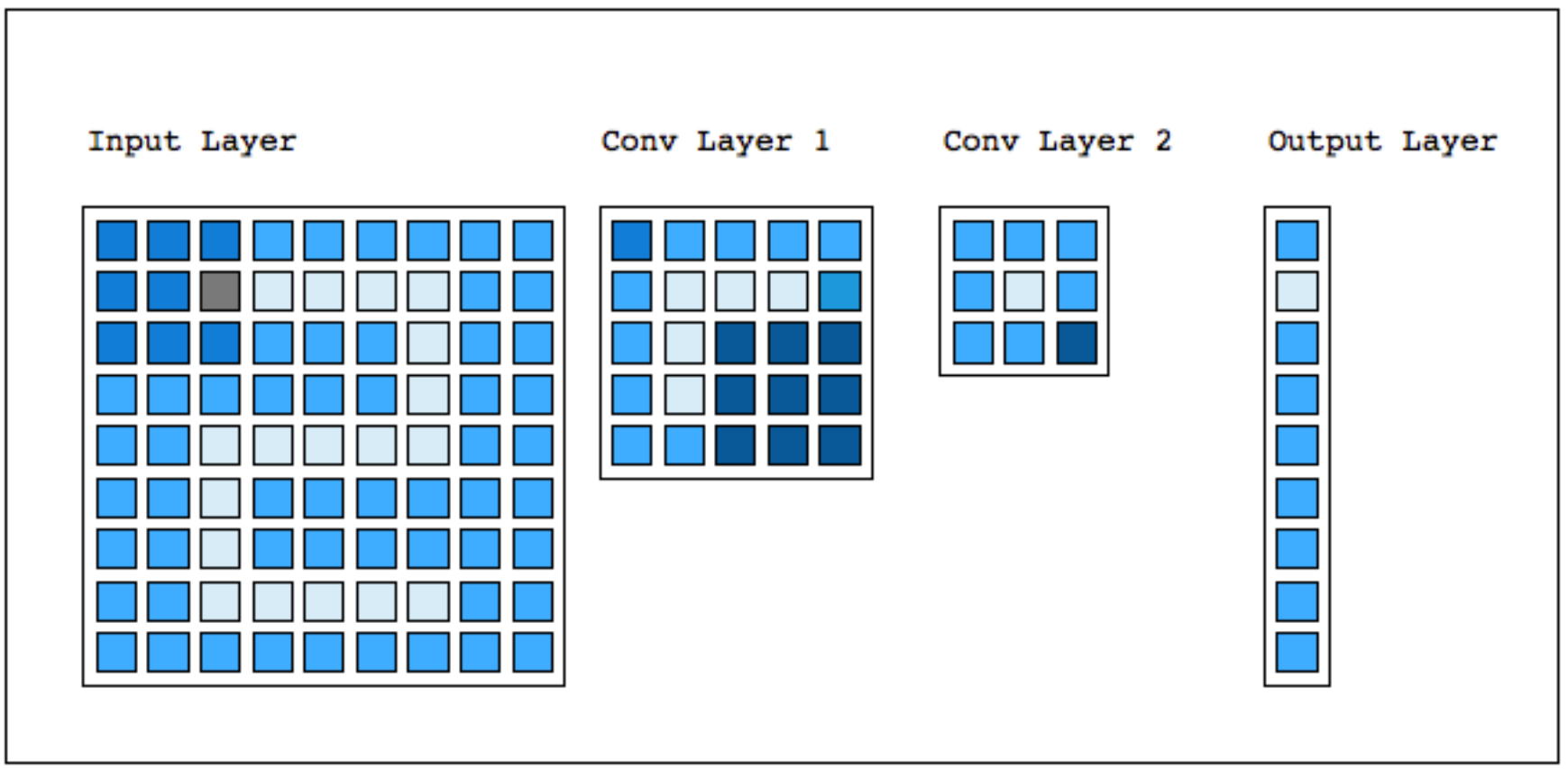
To achieve better results in image recognition tasks deeper networks are needed. And, ideally, they need to be capable of running convolutional layers. Hence, I set out to add these features, on my continous journey into the world of *machine learning*. :)

Convolutional Networks

Convolutional layers are a different beast than normal fully connected layers. Instead of each node in a layer connecting to *all* nodes in the previous layer, it connects to

only *some* of them. The selection of which nodes it connects to is defined by a quadratic filter that moves over the previous layer. The geolocation of each node, i.e. the horizontal and vertical proximity to its neighbors, is thereby taken into account which is crucial for image recognition.

The second major difference of convolutional layers is the fact that weights are shared between nodes. This siginificantly reduces the network's complexity, i.e. the number of weights or parameters that need to be trained, as well as its memory requirements.



This post assumes that you understand the basic functionality of a convolutional network. If you don't I strongly recommend reading Stanford's CS231n Convolutional Neural Networks for Visual Recognition by Andrej Karpathy or, more hardcore, Yann Lecun's LeNet-5 Convolutional Neural Networks first.

So, now let's jump into the code and start with the network's overall design.

Network Architecture

More than anything else did the introduction of convolutional layers influence the design of the network. Previously, my network structure consisted of layers, nodes and weights. Now, I added 2 additional concepts: `Columns` and `Connections`. So in total, the neural network consists of 5 data structures.

The Network

The network structure serves as the overall *container* for the whole network. It defines the *learning rate* and information about number and location of all weights in the network. And, most importantly, it includes a variable-sized array of layer structures which contain the network nodes.

```
struct Network{
    ByteSize size;                  // actual byte size of thi
    double learningRate;            // factor by which connect
    int weightCount;                // number of weights in th
    Weight *weightsPtr;             // pointer to the start of
    Weight nullWeight;              // memory slot for a weigh
    int layerCount;                 // number of layers in the
    Layer layers[];                 // array of layers (of dif
};
```

A Network Layer

The `Layer` structure contains all the information related to a specific layer in the network, including a pointer to its weights and a variable-sized array of all columns inside this layer.

It's worth mentioning here that in my design, the input layer is counted as one of the layers. Other definitions may vary. Yann LeCun's collection of MNIST results, for example, defines a linear classifier as a 1-layer-NN, and networks with exactly 1 hidden layer as 2-layer-NNs. In my design these count as 2-layer-NN and 3-layer-NN respectively.

```
struct Layer{
    int id;                         // index of this layer in
    ByteSize size;                  // actual byte size of thi
    LayerDefinition *layerDef;      // pointer to the definiti
    Weight *weightsPtr;             // pointer to the weights
    int columnCount;                // number of columns in th
    Column columns[];               // array of columns
};
```

The second point worth mentioning here is that I keep the definition of the layer model outside of the `Layer` structure in a separate `LayerDefinition` structure (more on that below) and add an equivalent pointer reference to the `Layer`.

A Network Column

What is a network column and why are they needed? In my previous design, a network layer directly contained the network nodes that were all aligned flat in a 1-dimensional vector. A MNIST image, for example, has 28 x 28 pixels and the respective network layer thus consists of 784 nodes that are all aligned in a single row.

In image recognition, though, the exact location of a pixel inside the image matters. Convolutional networks therefore build connections to neighboring nodes that are located within a defined region, the so called *filter*.

Example: A convolutional network node may want to create connections to a 3 x 3 area (the *filter*) of 9 neighboring nodes (pixels) located at the top left of the image. If all pixels of the image are numbered from 0..783 than this area can be easily defined as nodes [0,1,2,28,29,30,56,57,58]. We essentially use a 1-dimensional vector to include some 2-dimensional information using simple algebra.

So far, so good. The critical point that triggered a design change is that images are not 2-dimensional but 3-dimensional. What? Yes. Convolutional networks treat their input, be it an image in the input later or the output from a previous layer, as 3-dimensional.

Where does the 3rd dimension come from? … Color. In a simple RGB image, each pixel is composed of 3 values: a red value, a green value and a blue value. The MNIST images are actually not color but grey-scale. Thus, if it was only about MNIST we wouldn't need the 3rd dimension.
But the idea here is to build an architecture as flexible as possible that can handle all kinds of data sets.

Columns vs Feature Maps

Adding a 3rd dimension of nodes to the network, I faced 2 design options: feature maps or columns. Intuitively, I wanted to slice the image (or the input of a previous layer for that matter) **horizontally** which creates what the convolutional network theory sometimes refers to as *feature maps*. So in the example of an RGB image, instead of having a single 1-dimensional [0..783] vector of 784 nodes, you'd have an array of 3 *feature maps* and each would consist of 784 nodes.

Alternatively, if you slice the image (or the input of a previous layer for that matter) **vertically** you end up with `Columns`. The network layer then consists of an array of

784 `Columns` each consisting of 3 nodes.

Conceptually, both designs are equivally acceptable and feasible to implement. The fact that I ended up chosing the latter may have been related to my previous study of Hierarchical Temporal Memory (HTM) theory where columns are an intrinsic element. This design overall also more closely resembles the design of the brain.

```
struct Column{
    ByteSize size;                  // actual byte size of this st
    int maxConnCountPerNode;        // maximum number of connectic
    int nodeCount;                  // number of nodes in this col
    Node nodes[];                   // array of nodes
};
```

A Network Node

A column contains a number of network nodes. In non-convolutional layers the number of nodes per column is, by definition, always 1. I call this the *depth* of the layer. Thus non-convolutional layers always have a *depth* of 1 which means the number of columns equals the number of nodes.

```
struct Node{
    ByteSize size;                  // actual byte size of this st
    Weight bias;                    // value of the bias weight of
    double output;                  // result of activation functi
    double errorSum;                // result of error back propag
    int backwardConnCount;          // number of connections to th
    int forwardConnCount;           // number of connections to th
    Connection connections[];       // array of connections
};
```

The Connections

The 2nd new concept being added to the network model is the introduction of `Connections`. In my previous network `Weights` were attached directly to a `Node` because each node had exactly 1 weight. And the *target* `Node` to which a certain node is connectd to could be easily derived from the weight's id. For example, the 1st weight of each node in the hidden layer is applied to the 1st node in the input layer. Thus, these 2 nodes are connected. The 2nd weight of each node in the

hidden layer is applied to the 2nd node in the input layer. And again, these 2 nodes become connected.

That's simple. For convolutional networks, things get more complicated. First, weights are shared, i.e. the same weight will be used from (i.e. applied to) multiple nodes. Second, each node in the hidden layer is connected to different nodes in the previous layer. Therefore, we cannot simply assume weight 1 connects to node 1, weight 2 connects to node 2, etc. Instead, for each node in the hidden (convolutional) layer we need to keep track of what nodes in the previous layer (= *target* `Nodes`) it connects to.

Looking at biology, I introduced a new object `Connections` (similar to synapses) to the model. A connection is a structure simply storing 2 pointers: a pointer to a target node and a pointer to a weight.

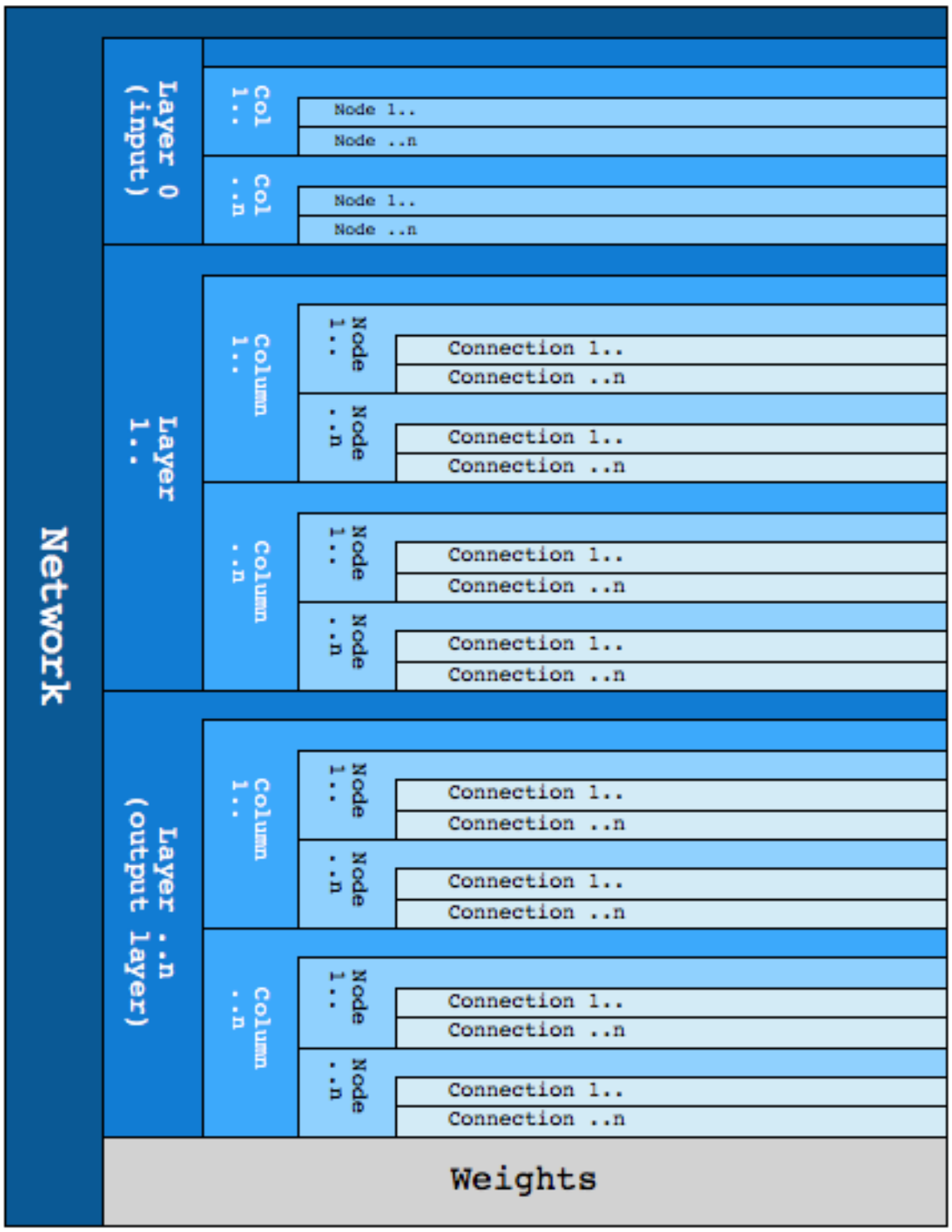
```
struct Connection{
    Node *nodePtr;                  // pointer to the target node
    Weight *weightPtr;              // pointer to a weight that is
};
```

The Weights Block

The most important component of a neural network is still missing in all of the above: the weights. The `Connection` structure only contains a pointer to a weight. But where is the actual weight?

The answer is I removed the weights from the nodes and instead put all together in a *weights block* located at the end of the `Network` structure, just after the last `Layer`. This design has several advantages: first, the weights are kept together with the rest of the network inside the same memory block. Second, the separation of weights from the nodes allows for convenient weight sharing.

The following drawing shows how all of the above components fit together to create the `Network` structure.



Network Definition

To build a network one first has to define a model. How many layers shall the network have, how may nodes are inside each layer, etc. To do so I introduced a new structure called `LayerDefinition` that holds the key definition parameters of each layer and that will be attached to the respective layer structure via a pointer.

```
struct LayerDefinition{
    LayerType layerType;         // what kind of layer is this
    ActFctType activationType;   // what activation function is
    Volume nodeMap;              // what is the width/height/de
    int filter;                  // size of the filter window (
};
```

To create a network you first create a variable number of these `LayerDefinitions`

and then pass them all together into the `setLayerDefinitions` function which returns an array pointer for easier reference throughout the rest of the code.

```
// Define how many layers
int numberOfLayers = 4;

// Define details of each layer
LayerDefinition inputLayer = {
    .layerType        = INPUT,
    .nodeMap          = (Volume){.width=MNIST_IMG_WIDTH, .h
};

LayerDefinition hiddenLayer1 = {
    .layerType        = CONVOLUTIONAL,
    .activationType   = RELU,
    .nodeMap          = (Volume){.width=13, .height=13, .de
    .filter           = 5
};

LayerDefinition hiddenLayer2 = {
    .layerType        = CONVOLUTIONAL,
    .activationType   = RELU,
    .nodeMap          = (Volume){.width=6, .height=6, .dept
    .filter           = 3
};

LayerDefinition outputLayer = {
    .layerType        = OUTPUT,
    .activationType   = RELU,
    .nodeMap          = (Volume){.width=10}
};

LayerDefinition *layerDefs = setLayerDefinitions(numberOfI
```

Above example defines the model for a 4-layer network, consisting of 2 hidden convolutional layers in addition to the input and the output layer. The 1st

convolutional layer is defined as having 13 x 13 nodes over 5 feature maps, resulting in 845 nodes in total. The 2nd convolutional layer is defined as having 6 x 6 nodes over 5 feature maps, resulting in 180 nodes in total.

I added some code to output the network's characteristics to the console for easier reference.

```
-------------------------------------------------------------------------------------------------------
Layer Index                    |       0       |       1       |       2       |       3       ||     TOTAL     |
Layer Type                     |     INPUT     |  CONVOLUTIONAL |  CONVOLUTIONAL |    OUTPUT     ||               |
-------------------------------------------------------------------------------------------------------
Activation Function            |               |      RELU     |      RELU     |      RELU     ||               |
Image Matrix (width x height)  |    28 x  28   |    13 x  13   |    6 x   6    |    10 x   1   ||               |
Feature Maps (depth)           |       1       |       5       |       5       |       1       ||               |
Filter Size                    |               |     5 x 5     |     3 x 3     |               ||               |
Stride                         |               |       2       |       2       |               ||               |
-------------------------------------------------------------------------------------------------------
Number of Nodes                |         784   |         845   |         180   |          10   ||       1,819   |
Number of Connections          |           0   |      21,125   |       8,100   |       1,800   ||      31,025   |
Number of Weights              |           0   |         125   |         225   |       1,800   ||       2,150   |
Memory Size (bytes)            |      43,944   |     982,944   |     166,216   |      29,400   ||   1,239,752   |
-------------------------------------------------------------------------------------------------------
```

You can see that the network resulting from this model has a total of 2,150 weights and only uses 1.2 MB of memory.

Layer Types

The code currently supports the following *types* of layers:

```
typedef enum LayerType {INPUT, CONVOLUTIONAL, FULLY_CONNECTED,
```

The idea here is that based on the type of layer certain rules are applied, for example, the connections between nodes are created differently.

Moving forward I want to expand this further to include other layer types, for example Hierarchical Temporal Memory (HTM) layers for sequentiell learning.

Activation Function

The code currently supports below 3 activation functions:

```
typedef enum ActFctType {SIGMOID, TANH, RELU} ActFctType;
```

Each layer can define its own activation function which is applied during feed forward (*activation*) as well as during error *back propagation*. In theory, you can design a network using different activation functions for different layers. In practive, however, this does not improve the network performance.

It's also important to note that other parameters, most significantly the *learning rate*, depend on what activation function is used. Hence, both of these two hyper-parameters should always be considered in tandem.

Create the Network

Once the network and its layers have been defined via the above *setLayerDefinitions* function you can create the actual network object via the `createNetwork` function

```
Network *nn = createNetwork(numberOfLayers, layerDefs);
```

which automatically allocates the required memory for this network, initializes its internal structures (see below) and returns a pointer for reference.

```
Network *createNetwork(int layerCount, LayerDefinition *layerD

    // Calculate network size
    ByteSize netSize = getNetworkSize(layerCount, layerDefs);

    // Allocate memory block for the network
    Network *nn = (Network*)malloc(netSize);

    // Set network's default values
    setNetworkDefaults(nn, layerCount, layerDefs, netSize);

    // Initialize the network's layers, nodes, connections and
    initNetwork(nn, layerCount, layerDefs);

    // Init all weights -- located in the network's weights bl
    initNetworkWeights(nn);
```

```
    return nn;
}
```

Calculate the Network's Memory

To create the network we first need to calculate how much memory it needs. Obviously, this depends on many different parameters, such as the number of layers, the number of nodes per layer, the size of the filter in a convolutional layer, etc.

```
ByteSize getNetworkSize(int layerCount, LayerDefinition *layer

    ByteSize size = sizeof(Network);

    for (int i=0; i<layerCount; i++){
        ByteSize lsize =getLayerSize(layerDefs+i);
        size += lsize;
    }

    // get size of weight memory block (located within network
    ByteSize weightBlockSize = getNetworkWeightBlockSize(layer

    // add weight block size to the network
    size += weightBlockSize;

    return size;
}
```

To calculate the network size, we need to calculate each layer's size which in return requires to calculate each column's size which depends on each node's size which depends on the number of connections, … and so on. You get the idea. Sounds complicated at first, but it's actually pretty straight forward. So, I'm going to skip a review of all the sub functions here. You can review them in the code (see link below).

Initialize the Network

In the above `createNetwork` function we saw that in addition to setting some default values for the network, we need to *initialize* the network and set random weights.

Now, what do I mean by *initializing* the network? Don't forget, until now the network is simply a block of memory with some unknown content inside. We need to insert our desired structure into this memory block so that it mirrors our design as outlined above: a network that holds layers that hold columns that hold nodes that hold connections which point to other nodes and weights. Let's see how this is done.

```
void initNetwork(Network *nn, int layerCount, LayerDefinition

    // Init the network's layers including their backward conn
    // Backward connections point to target nodes in the PREVI
    // (i.e. during calculating node outputs = node activation
    for (int l=0; l<layerCount; l++) initNetworkLayer(nn, l, l

    // Init the network's forward connections
    // Forward connections point to target nodes in the FOLLOW
    // and are used during BACK PROPAGATION (to speed-up calcu
    for (int l=0; l<layerCount; l++) initNetworkForwardConnect

}
```

First, we loop through all layers and initialize them one by one. After that, we loop again to set up each layer's forward connections. The reason why we need 2 loops here is that in order for the `initNetworkForwardConnections` function to work, each layer's following layer must have been initialized already.

The background is simple: `initNetworkForwardConnections` checks which nodes in the following layer point back to a node in this layer and then creates forward connections mirroring these backward connections. Obviously, if the following layer hadn't been setup yet, there wouldn't be any forward connections in this layer. Hence, 2 loops.

Now, let's see what actually happens inside the `initNetworkLayer` function:

```
void initNetworkLayer(Network *nn, int layerId, LayerDefinitic

    LayerDefinition *layerDef = layerDefs + layerId;

    // Calculate the layer's position by moving a single byte
    // by the total sizes of all previous layers
    uint8_t *sbptr1 = (uint8_t*) nn->layers;
    for (int l=0; l<layerId; l++) sbptr1 += getLayerSize(layer
    Layer *layer = (Layer*) sbptr1;

    // Calculate the position of this layer's weights block
    uint8_t *sbptr2 = (uint8_t*) nn->weightsPtr;
    for (int l=0; l<layerId; l++) sbptr2 += getLayerWeightBloc
    Weight *w = (Weight*) sbptr2;

    // Set default values for this layer
    layer->id                = layerId;
    layer->layerDef          = layerDef;
    layer->weightsPtr        = w;
    layer->size              = getLayerSize(layerDef);
    layer->columnCount       = getColumnCount(layerDef);

    // Initialize all columns inside this layer
    initNetworkColumns(nn, layerId);

}
```

In order to *initialize* a layer, we first need to know where each layer starts. Remember, since each layer has a different size, I cannot use a simple array reference such as `network.layer[1]` to locate a layer. Instead, I point a *single byte pointer* to the start of the network's layers *nn->layers* and move it forward by exactly the size of each layer.

Then, I define for each layer some basics such as its `id`, its `weightPtr`, etc., and add a reference to its original `LayerDefinition`. This will become handy because

througout the code we will need to access a layer's definition quite often.

Now, we've setup the *head* of the layer. What's missing is the structure underneath, i.e. the columns inside this layer. So that's what's next.

```
void initNetworkColumns(Network *nn, int layerId){

    Layer *layer = getNetworkLayer(nn, layerId);

    int backwardConnCount = getNodeBackwardConnectionCount(lay
    int forwardConnCount  = getNodeForwardConnectionCount(laye

    ByteSize columnSize = getColumnSize(layer->layerDef);

    // Init all columns attached to this layer
    for (int c=0; c<layer->columnCount; c++){

        // Set pointer to the respective column position (usir
        uint8_t *sbptr = (uint8_t*) layer->columns;
        sbptr += c * columnSize;

        Column *column = (Column*) sbptr;

        // Set default values of a node
        column->size     = columnSize;
        column->nodeCount= layer->layerDef->nodeMap.depth;
        column->maxConnCountPerNode = backwardConnCount+forwar

        // Initialize all nodes of a column
        initNetworkNodes(nn, layerId, c);

    }

}
```

Again, the same logic as above applies. Since the size of a column structure is variable I use a single byte pointer to determine the starting point of each column

inside this layer. Once I've done that, I continue to intitialize all of the *nodes* inside this column.

```
void initNetworkNodes(Network *nn, int layerId, int columnId){

    Layer *thisLayer    = getNetworkLayer(nn, layerId);
    Layer *prevLayer    = getNetworkLayer(nn, layerId-1);
    Column *column      = getLayerColumn(thisLayer, columnId);
    ByteSize nodeSize   = getNodeSize(thisLayer->layerDef);

    uint8_t *sbptr = (uint8_t*) column->nodes;

    // Create a vector containing the ids of the target column
    Vector *filterColIds = createFilterColumnIds(thisLayer, co

    // Init all nodes attached to this column
    for (int n=0; n<column->nodeCount; n++){

        // Set pointer to the respective node position (using
        Node *node = (Node*) sbptr;
        sbptr += nodeSize;

        // Reset node's defaults
        setNetworkNodeDefaults(thisLayer, column, node, &nn->r

        // Initialize backward connections of fully-connected
        if (thisLayer->layerDef->layerType==FULLY_CONNECTED ||

            int nodeId = (columnId * column->nodeCount) + n;

            // @attention When calculating the weightsId, only
            int layerWeightsId = nodeId * getNodeBackwardConne
            Weight *nodeWeight = thisLayer->weightsPtr + layer

            initNetworkBackwardConnectionsFCNode(node, prevLay
        }
```

```
        // Initialize backward conections of convolutional lay
        if (thisLayer->layerDef->layerType==CONVOLUTIONAL){
            // @attention Nodes on the same level share the sa
            initNetworkBackwardConnectionsConvNode(node, n, th
        }

    }

    free(filterColIds);
}
```

Create Backward Connections

This function is central to the network's initialization process. In addition to setting some default values for all nodes, it's preparing the creation of backward connections of each node to the previous layer. The way in which these backward connections are constructed is obviously different for *fully connected* and for *convolutional* layers. Hence there are 2 different sub functions, `initNetworkBackwardConnectionsFCNode` for fully connected nodes and `initNetworkBackwardConnectionsConvNode` for convolutional nodes.

Backward Connections for Fully Connected Layers

Let's look at the easier of the 2 first, the *fully connected* node:

```
void initNetworkBackwardConnectionsFCNode(Node *thisNode, Laye

    int connId = 0;
    int nodeWeightId = 0;

    ByteSize columnSize = getColumnSize(prevLayer->layerDef);

    uint8_t *sbptr_column = (uint8_t*) prevLayer->columns;

    for (int col=0; col<prevLayer->columnCount;col++){

        Column *column = (Column *)sbptr_column;
```

```
        uint8_t *sbptr_node = (uint8_t*) column->nodes;

        for (int n=0; n<column->nodeCount; n++){

            Connection *conn = &thisNode->connections[connId];

            conn->weightPtr = nodeWeightPtr + nodeWeightId;

            conn->nodePtr = (Node *)sbptr_node;

            sbptr_node += getNodeSize(prevLayer->layerDef);

            connId++;
            nodeWeightId++;
        }
        sbptr_column += columnSize;
    }
}
```

The function loops through the previous layer's nodes and creates a connection from the current node to **each** node in the previous layer (hence **fully** connected). Remember, since nodes are structured in columns we need to loop through the columns first and then through the nodes inside each column.

So far so good. That was the easier of the two. Now, let's look at how the wiring (building connections) works for *convolutional* nodes.

Backward Connections for Convolutional Layers

As outlined above, a convolutional node is connected to a selected group of neighboring nodes, located within a quadratic region of `filter` size. This quadratic region (the filter) needs to be calculated. It depends on a number of parameters and changes as the filter is moved across the target layer.

Example: Let's say our network consists of 3 layers, the input layer containing the MNIST image, a convolutional layer and the output layer. We know that the MNIST image has 28*28 pixels, hence the input layer has 784 columns. Each column only has 1 node (the `depth` of the input layer is 1) therefore there are also exactly 728 nodes. And let's assume our convolutional layer has dimensions of [24 * 24 * 10], i.e.

there are 24 * 24 = 576 columns and each column has 10 nodes. And let's assume we defined a `filter` of 5 for this layer.

Now we start wiring up the 1st node in the convolutional layer. We need to create connections to a region of 5 x 5 (`filter` * `filter` size) columns located at the top left of the target (in this case = input) layer. The respective ids of these columns would be:

```
[   0,   1,   2,   3,   4,
   28,  29,  30,  31,  32,
   56,  57,  58,  59,  60,
   84,  85,  86,  87,  88,
  112, 113, 114, 115, 116]
```

Then we move on to the 2nd node in the convolutional layer. Again, we create connections to a 5 x 5 region, but the region now moved to right along with the convolutional (*source*) node itself. Hence, the target column ids of the 2nd node in the convolutional layer would be:

```
[   1,   2,   3,   4,   5,
   29,  30,  31,  32,  33,
   57,  58,  59,  60,  61,
   85,  86,  87,  88,  89,
  113, 114, 115, 116, 117]
```

Above calculation is done by the `calcFilterColumnIds` function below. I trust you'll be able to walk through it, using the logic introduce in the example above.

```
void calcFilterColumnIds(Layer *srcLayer, int srcColId, Layer

    int srcWidth  = srcLayer->layerDef->nodeMap.width;
    int tgtWidth  = tgtLayer->layerDef->nodeMap.width;
    int tgtHeight = tgtLayer->layerDef->nodeMap.height;

    int filter = srcLayer->layerDef->filter;
    int stride = calcStride(tgtWidth, filter, srcWidth);
```

```
int startX = (srcColId % srcWidth) * stride;
int startY = floor((double)srcColId/srcWidth) * stride;
int id=0;

for (int y=0; y<filter; y++){

    for (int x=0; x<filter; x++){

        int colId = ( (startY+y) * tgtWidth) + (startX+x);

        // Check whether target columnId is still within t
        // If NOT then assign a dummy ID ("OUT_OF_RANGE")
        if (
            (floor(colId / tgtWidth) > (startY+y)) ||   //
            (colId >= tgtWidth * tgtHeight)             //
            ) colId = OUT_OF_RANGE;

        filterColIds->vals[id] = colId;
            id++;
        }
    }
}
```

Once we calculated this vector of `targetColumnIds` we pass it into our `initNetworkBackwardConnectionsConvNode` function which then creates connections from the source node to the target nodes specified in the vector.

```
void initNetworkBackwardConnectionsConvNode(Node *node, int sr

    int filterSize = filterColIds->count;
    int tgtDepth   = targetLayer->layerDef->nodeMap.depth;

    for (int posInsideFilter=0; posInsideFilter<filterSize; po

        int targetColId = (int)filterColIds->vals[posInsideFil
```

```
        for (int tgtLevel=0; tgtLevel<tgtDepth; tgtLevel++){

            Connection *conn = &node->connections[ (tgtLevel*f

            if (targetColId!=OUT_OF_RANGE){

                int weightPosition = (srcLevel*(tgtDepth*filte

                Weight *tgtWeight = srcLayerWeightPtr + weight

                Node *tgtNode = getNetworkNode(targetLayer, ta

                conn->nodePtr    = tgtNode;
                conn->weightPtr = tgtWeight;
            }
            else {
            // if filter pixel is out of range of the target r
            // this kind of pointer needs to be captured later
            conn->nodePtr    = NULL;
                conn->weightPtr = nullWeight;
                }
            }
        }
}
```

Depending on how many nodes we defined in our convolutional layer, some of the nodes in the target (=previous) layer are purposely skipped. This is called the `stride`. A `stride` of 2 means that every other node in the target layer is skipped. The idea is to *downsize* the original image layer by layer, and have each following layer represent some higher level feature of the previous layer.

```
int calcStride(int tgtWidth, int filter, int srcWidth){
  return ceil(((double)tgtWidth - filter)/(srcWidth-1));
}
```

Create Forward Connections

What are forward connections and why do we need them? Let's revisit some of the

above. A connection links 2 nodes and applies a weight during the *feed forward* process. Then, during the *back propagation* process, the same connection is traversed in opposite direction to pass the partial error of a node back to the connected node in the previous layer. It's the latter that triggered the need for *forward* connections.

How? C pointers are one-directional. Variable A points to variable B but variable B does not know who points to it. Therefore, I cannot simply traverse a connection between nodes in opposite direction. So instead, I create additional connections from the target back to the source.

Both types of connections, backward and forward, serve different purposes:

```
Backward connections are used during feed forward.
```

```
Forward connections are used during back propagation.
```

Yes, that's no typo. The above may sound counter-intuitive at first, but makes sense when you think about it.

During *feed forward* you need to calculate the output of a node. To do so, you need to loop through all of its **incoming** (=backward) nodes to calculate the vector product and apply an activation function to the output.

During *back propagation* you need to calculate the error of a node. To do so, you need to loop through all its **outgoing** (=forward) nodes to obtain the partial error from this node and apply a derivative function.

So, here's the function for initializing the network's forward connection:

```
void initNetworkForwardConnections(Network *nn, int layerId) {

    // Skip the first=INPUT and the last=OUTPUT layer because
    if (layerId==0 || layerId==nn->layerCount-1) return;

    Layer *thisLayer = getNetworkLayer(nn, layerId);
    Layer *nextLayer = getNetworkLayer(nn, layerId+1);

    for (int c=0; c<thisLayer->columnCount; c++){
```

```
        Column *column = getLayerColumn(thisLayer, c);

        for (int n=0; n<column->nodeCount; n++){
            Node *node = getColumnNode(column, n);
            initNetworkForwardConnectionsAnyNode(node, nextLay
        }
    }
}
```

It simply loops through all layers and their respective columns and calls the `initNetworkForwardConnectionsAnyNode` function which does the actual wiring:

```
void initNetworkForwardConnectionsAnyNode(Node *thisNode, Laye

    int maxForwardConnCount = thisNode->forwardConnCount;
    int forwardConnStart = thisNode->backwardConnCount;
    int forwardConnCount = 0;

    for (int o=0;o<nextLayer->columnCount;o++){
        for (int n=0; n<nextLayer->columns[0].nodeCount; n++){

            Node *nextLayerNode = getNetworkNode(nextLayer,o,
            for (int c=0;c<nextLayerNode->backwardConnCount;c+

                // If the connection of the node in the next l
                // then store this nextNode as a target in the
                // and point the connection to the same weight
                if (nextLayerNode->connections[c].nodePtr == t
                    thisNode->connections[forwardConnStart + f
                    thisNode->connections[forwardConnStart + f
                    forwardConnCount++;
                    if (forwardConnCount>maxForwardConnCount)
                        printf("Maximum forward connections ex
                        exit(1);
                    }
                }
            }
```

```
        }
    }
    thisNode->forwardConnCount = forwardConnCount;
}
```

Initialize Weights

Now that we've fully setup all of the connections between the nodes and from the nodes to their weights, we still need to initialize the weights. Remember, all weights are located in a big weight block at the end of the `Network` structure.

All we need to do is initialize each weight with a random value from 0-1. For better performance I found that it helps to reduce the value to a random number smaller than 0.5 or 0.4 and also to make every other weight negative.

```
void initNetworkWeights(Network *nn){

    // Init weights in the weight block
    for (int i=0; i<nn->weightCount; i++){
        Weight *w = &nn->weightsPtr[i];
        *w = 0.4 * (Weight)rand() / RAND_MAX;    // multiplying
        if (i%2) *w = -*w;                       // make half c
    }

    // Init weights in the nodes' bias
    for (int l=0; l<nn->layerCount;l++){
        Layer *layer = getNetworkLayer(nn, l);
        for (int c=0; c<layer->columnCount; c++){
            Column *column = getLayerColumn(layer, c);
            for (int n=0; n<column->nodeCount; n++){
                // init bias weight
                Node *node = getColumnNode(column, n);
                node->bias = (Weight)rand()/(RAND_MAX);
                if (n%2) node->bias = -node->bias;  // make ha
            }
        }
    }
}
```

Train the Network

Once the network has been fully setup and initialized, we can train it in exactly the same manner as described in my previous post:

```
1. read an image
2. feed it into the input layer of the network
3. feed all layers forward applying an activation function eac
4. backpropagate the error through all layers and update its w
```

To obtain some indication of the network's progress during the training process we can count the number of correct classifications versus the total number of images and calculate an ongoing accuracy rate.

```
void trainNetwork(Network *nn){

    // open MNIST files
    FILE *imageFile, *labelFile;
    imageFile = openMNISTImageFile(MNIST_TRAINING_SET_IMAGE_FI
    labelFile = openMNISTLabelFile(MNIST_TRAINING_SET_LABEL_FI

    int errCount = 0;

    // Loop through all images in the file
    for (int imgCount=0; imgCount<MNIST_MAX_TRAINING_IMAGES; i

        // Reading next image and its corresponding label
        MNIST_Image img = getImage(imageFile);
        MNIST_Label lbl = getLabel(labelFile);

        // Convert the MNIST image to a standardized vector fo
        Vector *inpVector = getVectorFromImage(&img);
        feedInput(nn, inpVector);

        // Feed forward all layers (from input to hidden to ou
        feedForwardNetwork(nn);
```

```
        // Back propagate the error and adjust weights in all
        backPropagateNetwork(nn, lbl);

        // Classify image by choosing output cell with highest
        int classification = getNetworkClassification(nn);
        if (classification!=lbl) errCount++;
    }
    // Close files
    fclose(imageFile);
    fclose(labelFile);
}
```

Once the network has been trained on the the 50,000 images in the training set (no distinction between training and validation set is made), we stop training and run the network on the 10,000 images in the testing set.

Test the Network

The process during testing is exactly the same as during training, the only difference is that we switch off the *learning*, i.e. we don't back propagate the error and don't update any weights.

```
void testNetwork(Network *nn){

    // open MNIST files
    FILE *imageFile, *labelFile;
    imageFile = openMNISTImageFile(MNIST_TESTING_SET_IMAGE_FII
    labelFile = openMNISTLabelFile(MNIST_TESTING_SET_LABEL_FII

    int errCount = 0;

    // Loop through all images in the file
    for (int imgCount=0; imgCount<MNIST_MAX_TESTING_IMAGES; im

        // Reading next image and its corresponding label
        MNIST_Image img = getImage(imageFile);
        MNIST_Label lbl = getLabel(labelFile);
```

```
        // Convert the MNIST image to a standardized vector fo
        Vector *inpVector = getVectorFromImage(&img);
        feedInput(nn, inpVector);

        // Feed forward all layers (from input to hidden to ou
        feedForwardNetwork(nn);

        // Classify image by choosing output cell with highest
        int classification = getNetworkClassification(nn);
        if (classification!=lbl) errCount++;
    }
    // Close files
    fclose(imageFile);
    fclose(labelFile);
}
```

Output Classification

In case you wondered how the *classification* of an image in the output layer is done: I'm not using a *softmax* or similar function. Instead, I'm simply comparing all outputs of the 10 nodes in the output layer with each other and take the highest. :-)

```
int getNetworkClassification(Network *nn){

    Layer *l = getNetworkLayer(nn, nn->layerCount-1);   // get

    Weight maxOut = 0;
    int maxInd = 0;

    for (int i=0; i<l->columnCount; i++){
        Node *on = getNetworkNode(l,i,0); // the output layer
        if (on->output > maxOut) {
            maxOut = on->output;
            maxInd = i;
        }
    }
    return maxInd;
```

```
}
```

That's it. Now that we've done all the coding, let's look at performance.

Network Performance

To assess the performance of the network let's look at its accuracy as well as its speed. Here's a summary of the different network designs that I tried and their respective best results:

```
TEST RESULTS

| Network      | # of Nodes      | Act.Fct | L-Rate | # Trains |
| ------------ |-----------------|---------|--------|----------|
| 1-l   FC NN |       10 output | SIGMOID | 0.0125 |  900,000 |
| 2-l   FC NN |     300 hidden  | SIGMOID | 0.0700 |  120,000 |
| 3-l   FC NN | 500+150 hidden  | SIGMOID | 0.0050 |  180,000 |
| 3-l ConvNet | 13x13x5, 6x6x5  |  RELU   | 0.0004 |  525,000 |

 * total run-time in seconds, including training and testing,
** best accuracies listed on http://yann.lecun.com/exdb/mnist/
```

As you can see from the above my network is doing well for smaller models but not so much for larger, deeper structures. This is not due to wrong coding. It's because I haven't used the most efficient parameters during the training.

This leads us to the most difficult part of machine learning: How to find the hyper-parameters that achieve the best results?

Hyper-parameter Optimization

Surprisingly, optimizing hyper-parameters such as the learning rate or the number of nodes is still mostly a manual process. Based on my reading it seems that techniques such as *grid search* or *Bayesian Optimization* work equally good or bad than a *Random Search* does.

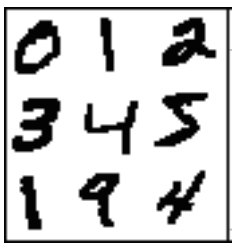
Therefore, I plan to dedicate my next post to the topic of network parameter fine

tuning and build my own optimization tool to get some bettter results from my network. Stay tuned! :)

Code & Documentation

As always you can find all the code for this exercise on my Github project page, including full code documentation.

Happy Hacking!



Written on February 12, 2016