Obligatory Exercise 3

Group 8 - Report

090832	Magnus Øverbø
130197	Adrián Alberdi Ainziburu
131724	Miranda Qorolli

Date:	April 18, 2014
Version:	1.0
Teacher:	Sule Yildirim Yayilgan
Page count:	
Word count:	N/A
Appendices:	0
Keywords:	Artificial Intelligence, graphs, network, neural network, machine learning

IMT3391 — Artificial Intelligence

Department of Computer Science and Media Technology
Gjøvik University College

Contents

1 Image Data

1.1 Image Set

The images we had to classify consisted of images representing the 26 upper case characters A to Z. For each character there was a set of 20 images with minute differences in appearance.

De images is in the form of grayscale images which uses the JPEG compression method for storage and file format. Which means that they have a specific compression algorithm, and file format.

Even though all images are the size 30x30 pixels, they will all have different sizes because of the compression algorithm. This means that we have to read images using an image library for C++ or pre-convert the images into a specific file format.

1.2 Conversion

To read the images in the software(C++) we tried different libraries which had image reading capabilities. Reading them as plain binary files would'nt have worked since they have embeded metadata and are compressed with the JPEG format.

Because of this we tried ImageMagick and OpenCV, but neither worked with any ease so we decided to use Python to convert them into a play text file with the images pixel values as text data.

So by creating the script to convert images, see appendix A.1, we were left with a single file holding one informative line with which character was depicted in the set and the output target values. This is then followed by 20 lines, each holding 100 decimal values depicting the pixel values of the image.

The script will open the pictures in ascending order starting with the images dipicting As, then Bs and so on, until Z. And for each of these characters it will read its 20 images straight away.

Once it has loaded the image, using Python Image Library(PIL), it will go through a 10x10 array which will create the new picture. Then it will pick out a 3x3 subset of the image, see figure 6, and average its value. Then it will add it to a list for printing. It also adds it to a new 30x30 image that depicts the scaled doen version for comparison, though it is still in 30x30px and the image data is only 10x10 scale..

1.2.1 Data Verification

To verify that the converted images was in fact correctly created, we took two approaches. First as well as grabbing the average value of the 3x3 subsets we created a scaled down picture set of the converted data. This served as a visual verification of our data conversion. The second verification method was to create a small Python script, see listing A.2 , that read the data from our converted file and displayed the images as a 10x10 array of symbols in the console.

1.3 In software

Since we decided to pre-convert the images into data files as described above we needed a custom datastructure to hold the data in memory while running the program. We created the following class which creates a long list of ”Data” objects where each objects contains the data for one image.

In the program there are two global variable directly linked to the Data structure. The global variable ”curTarget” and ”inData” which holds the current array of target values, and the current Data object which holds the needed image data.

1.3.1 Data structure

We've depicted the data structure in the UML figure 5.

fasit: This holds which character the image data represents, in upper case ASCII letters between A and Z.

target: This is a pointer to an integer array which will hold the target values. This is used when pushing it through the network to decide hwo much each output node is off target. The array size is the same as the number of output perceptrons.

values: This is a pointer to a float array, which holds all of the input values for the input layer. This is used to set the initial values of the input layer which then is transformed with an activation function. The size of the array is the number of input perceptrons.

next: This is a pointer to the next item in the list of Data objects. It is used for grabbing the next object in the list to perform some actions on.

2 Artificial Neural Network

An artificial neural network is constructed as shown in figure 2. Where one has the input layer on the bottom which takes a set of input values an propagates them to the next layer. This next layer is the first of one or more layers that propagates the input values through many weighted synapses which have been through the perceptrons activaiton function. After passing through the entire hidden layer, the values comes tho the output layer which will calculate the output value.

After this the software calculates the errors for each of the outputs and decides whether it has converged. All output nodes that has converged are marked and when all nodes has converged, the entire network has converged and should provide the desired output.

2.1 Perceptrons (Nodes)

The perceptron is the junction points in the network where the outputs of the previous layer is pushed into the new layer. The perceptron receives the weighted outputs of the previous layer and stores the sum of all its inputs.

The input is then pushed through the activation function of the perceptron, which generates the perceptrons output value. This output value is the value which is passed on to the next layer through the synapses. As it passes throught he synapses it's weighted and the resulting value is pushed to the linked perceptron.

2.1.1 Activation function

activation function is a mathematical operation that takes the net input and calculates an output value. The most used functions is the hyperbolic tan function and the Sigmoid function. The function which we use are the sigmoid function which generates a value between -1 and +1 from the net input value. It uses the function, $1 \div (1 + e^{-net})$ to generate the output value.

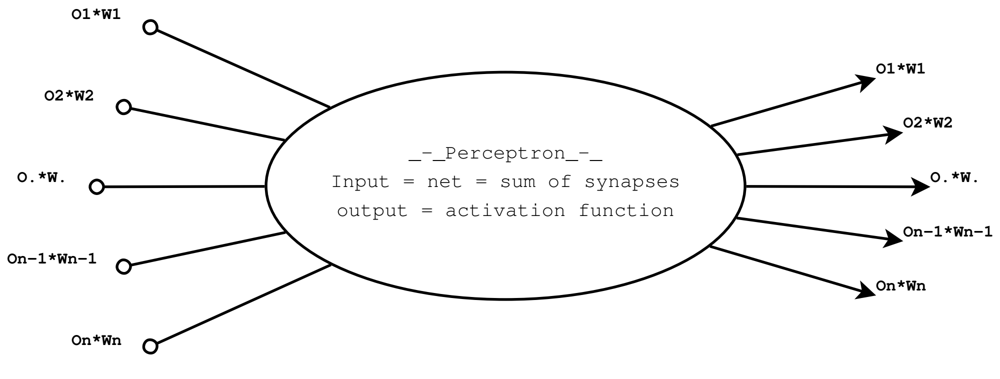


Figure 1: Illustration of a perceptron with n inputs and n outputs

2.1.2 Software representation

We have created a representation of a perceptron with the class ”Node”, which is depictedn in figure 4, using UML to describe it. It has a set of variables which holds the information and state of the node.

id	This is the integer id of the perceptron(node), and is only unique within the layer it is in. It is mostly used for identification, but also for making it easy to insert new values from the data structure, since it can retrieve its id from the value array.
input	This is a float value which represents the input value from either from the data set or the weighted outputs from the previous layer.
output	This holds the output value created by the activation function of the node. For most layers that is the sigmoid function, but also a custom function to decide the output from the input layer.
error	This holds the error returned from the error calculation function, as a part of the backpropagation algorithm. This is then used by the lower layer to adjust its weights linking to the layer.
***weight**	This is a pointer to the first element of a list containing the set of weights connecting it to the next layer.
***next**	This is a pointer to the next instance of a object in the list of Nodes(layer), the last instance in the list will point to ”NULL”.
convergence	This is a boolean value stating whether or not the Node has reached convergence as desired by the output.

2.2 Synapses (Links)

A synapse is the directed link between to perceptrons. It’s purpose is to weight the signal passing through it to give a better result to the output. As the neural network training progresses this weight changes to provide the best end output for the overall network. The change is made using the backpropagation algorithm, see chapter 3.

2.2.1 Software representation

We have created a representation of the synapses with the class ”Weight” which is depicted in figure 3, using UML to describe it. It has a set of variables to hold the information and make it possible to use te backpropagation algorithm. It also have a set of methods to help with retrieving the information in order to facilitate the algorithm.

The ”Weights” is a member of the perceptron and forms a list of all connections the perceptron has a connection to.

id	The id is an auto incremented counter that identifies which weight we are currently accessing in the list. Mostly used for identifying the weight/synapse during information display.
to	This variable is a pointer, which points to the perceptron(node) that the synapse(weight) connects to. This is used to grab a hold of the linked node when pushing the input value through the network and so on. And must be here since it is a feed forward network.

weight	This is a float value that holds the current weight value of the synapse. This is the weight that is used when propagating the input value through the network.
change	This is a float value that holds the value of the previous weight change. This value is decided when calculating the weight change during backpropagation. The change value is then multiplied by the momentum, a small decimal value.
next	This is a pointer to the next Weight in the list. The last element in the list will point to "NULL".

2.3 Construction

After a couple of tries we decided to create a fully connected ANN(Artificial Neural Network). This means that every perceptroon on the previous layer has a synapse to each perceptron on the next layer. As depicted in figure 2.

For our network we have created a general implementation which makes it possible to create a input layer, arbitrary amount of hidden layers and a output layer. The input and output layer can have arbitrary amounts of nodes, but the hidden layers must all have the same amount of nodes.

For our network we have created a network with 100 input perceptrons and 26 output perceptrons. The hidden layer is then chosen to have an arbitrary amount of nodes and layers so we could test different implementations.

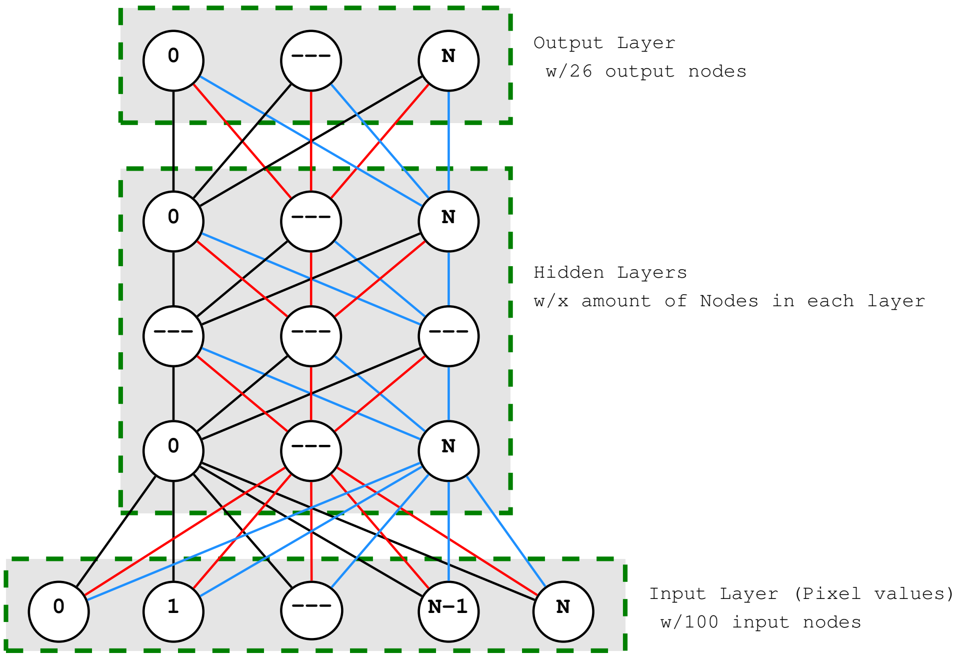


Figure 2: Illustration of a fully connected, neural network

The network construction starts by creating the input layer. It creates a list of Node objects that starts at the global "input" pointer, which makes it easy to grab a hold of and use later.

Then it does the same for each of the hidden layers and output layer.

Once this is finished we have a set of lists made up by Node objects, depicting perceptrons, but none of them are linked to eachother. This is the next step.

We start with the input list, and grab the global pointer which points to the first object of the list. Then we call a the Nodes local method, ”setWeight”. It takes one parameter, a pointer to a Weight object. This pointer should then point to a list of Weight objects that has links to the next layer of nodes.

This is done by calling it in the following manner:

```
input->setWeight( genWeights( NUM\_IN\_HIDDEN, hidden[0] ));
```

Here we call the genWeights funciton which takes a int and a list of perceptrons as input. It then generates a list of Weight-objects with random weights that links to the provided layer of nodes.

This way we create for each Node full connection to the next layer.

2.3.1 Input Layer

We chose to use a 100 input perceptrons because it provided a reasonable amount of accuracy in the images data to recreate the picture without loosing the information.

To set the input values we have a function that we call on the ”input” layer. ”setInputs” which then will call getValue on the global inData object, using its id as paramater. This enables it to consequently and easily retrieve the same index each time.

2.3.2 Hidden layers

There are no special functionality surrounding the hidden layer, except when we are updating weights, which is mostly done through the hidden layers.

2.3.3 Output Layer

The output layer consists of 26 perceptons because it makes it easy to see what charachters the input data is classified as. Since we want a value between 0 and for each output perceptron we can multiply by 100 and get the percentage of classification for the data being a specific character.

The output the tests yields is in the form below. This example says that it has classified the input data as 20% similar to a ’U’, 58% similar to a ’G’, 40% similar to ’D’, 90% similar to ’C’ and 20% similar to a ’B’. Because of this on can confident that it is a ’C’.

CHAR	id	Z	Y	X	W	V	U	T	....	H	G	F	D	C	B	A
C	4	0%	0%	0%	0%	0%	20%	0%	....	0%	58%	0%	40%	90%	20%	0%

3 Backpropagation Algorithm

3.1 Input propagation

Once the all input nodes has received their initial input value between 0 and 255. We performa a custom calculation to reduce this value. We tried the sigmoid function, but it returned too many ones since everything above 13 becomes 1.

We have tried a couple of different functions whichs is listed below.

- $O_{(net)} = (net \div 128) - 1$, this function creates a value between -1 and +1 relative to its original value. The value makes classification hard since large differences in original values will have small differences here.
- $O_{(net)} = ((net < 200)?1 : 0)$, this function says that if the value of the pixel is below 200 the output value should be considered as 1. This makes all dark areas classified as 1 and all light areas classified as 0. We've tried different variations of this, but 200 is an okay value.
- $O_{(net)} = 1 \div (1 + e^{-net})$, this sigmoid function yields almost all ones as output because after a given height it will allways yield 1 as the output.

After calculation the first output value we go through each node in the layer and call the pushForward method that goes through the nodes weight list and updates the linked nodes input value by appending the output value multiplied by the weight of the synapse.

When this is done we call the calcOutput method of the newly updated layer, which in return will calculate the output using the sigmoid function.

These to steps of pushForward to the next layer and the calculating the next layers new output is done for all layers until it has calculated the output for the output layer.

Once this is finished the next step is to calculate the errors.

3.2 Errors

3.2.1 Offset from target

To validate that the output we receive is within the allowed offset from the target we calculate offset from target. If this is within the threshold we set the converged state of the node to true. This signify that the node has reached the level of optimization we feel is nessecary.

3.2.2 Output Error

Along with the offset we calculatet he output error. This is the error value we use during weight correction later on. The error for the output layer is calculated using the following function for each of the output nodes.

$$\delta_i = (T_i - O_i) * O_i * (1 - O_i)$$

- δ_i is the error for the current node
- T_i is the target relative to the current node

- O_i is the output of for the current node

3.2.3 Hidden layer errors

For the hidden layers we have to calculate the errors in a different way. This method is used for all the hidden layers. First one has to summarize the weighted errors of all connected nodes, which is to take error in the end node and multiply it by the weight of the synapse.

The following function is used to calculate the error for the hidden layer.

$$\delta_i = O_i * ((1 - O_i) * \Sigma(\delta_k * \omega_n))$$

- δ_i The error for current node
- O_i Output from current node
- δ_k The error in the node from next layer
- ω_n The weight of the link to the node in the next layer
- $\Sigma(\delta_k * \omega_n)$ The weighted sum of the errors of connected nodes

3.3 Weight changes

3.3.1 Main update

The weights are updated from the last hidden layer and to the first layer, going one layer at the time. The formula for how much the weight is to change is the following. Where i is the node on the current layer and k is the node on the layer above. It also has to take into account the previous change of that occured, which is the momentum. The momentum of the previous change is added to the current change.

$$\Delta\omega_{i,k} = \eta * \delta_k * O_i + (\alpha * \Delta\omega_{prev})$$

- $_i$ – This is the id of the node on the current layer.
- $_k$ – This is the id of the node on the layer above, which the synapse is linking to.
- $\Delta\omega_{i,k}$ – This is the total value to change for the synapse in question.
- η – Learning rate, a fixed value of how much of the error is to be corrected.
- δ_k – The error of the node the synapse is linking to.
- O_i – The output for the current node.
- α – Momentum, a fixed value of how much of the previous change the current synapses weight should also be changed with.
- $\Delta\omega_{prev}$ – The previous weight change of the synapse.

3.3.2 The Next Steps

When all weights/synapses has been updated, the next step is to clean house. To avoid any erros due to residual data we go through all nodes and resets their non-persistant data, which is error, input, convergence and output.

This is to have correct input values since they are appended to the original value.

Then the next step is to choose a new data set to learn from. The dataset should be a completely different, otherwise the network will learn to identify the one dataset and then the other, forgetting the first in the meantime.

Because of this conundrum we are moving throug the dataset as follows. Say we have 10 images per character we are going to learn. We then start with A1, then B1, C1, and so on until Z1. Then we increment the image set counter and reset the character set. Then we learn A2, B2, C2, ..., Z2. and so on until we reach Zn. Then if the network has converged we can stop otherwise we restart and train the network on the data set once more.

Usually we train the data set around 20.000 times for the entire data set of input values. from A1 to Zn.

4 Neural Network Training and Testing

4.1 Training Procedure

1. Load image data into data structure.
2. Generate fully connected feed-forward neural network.
3. Generate initial weights(synapses) for the entire network.
4. Select data set to use as input for network.
5. Propagate input values through the network.
6. Calculate output layer errors.
7. Calculate errors for the hidden layers.
8. Fix the weights in the network, according to the errors.
9. Repeat from step 4, until reached convergence and/or run the learning procedure 200.000 times.
10. Save all the weights to file.
11. Exit the program gracefully.

4.2 Testing Procedure

1. Load image data into data structure.
2. Generate fully connected feed-forward neural network.
3. Generate initial weights(synapses) for the entire network using the saved file generated by the learning program.
4. Select data set to use as input to network.
5. Propagate input values through the network.
6. Classify the output information into reasonable data.
7. Display the generated classification data.
8. Repeat from step 4 until it has gone through each data set.
9. Exit program gracefully.

5 Performance

5.1 Same set testing

5.2 Learning Set 1-10

5.2.1 Set 1-10 - Same Set Test

#	Z	Y	X	W	V	U	T	S	R	Q	P	O	N	M	L	K	J	I	H	G	F	E	D	C	B	A

5.2.2 Set 11-20

5.3 Learning Set 11-20

5.3.1 Set 11-20 - Same Set Test

List the average values of the resulting data set

5.3.2 Set 1-10

List the average values of the resulting data set

6 Results

6.1

6.2

6.3

List of Figures

Code snippets and scripts

List of Tables

A Scripts and Code Snippets

A.1 Image Converter

Code A.1: Script to convert all images into a text file

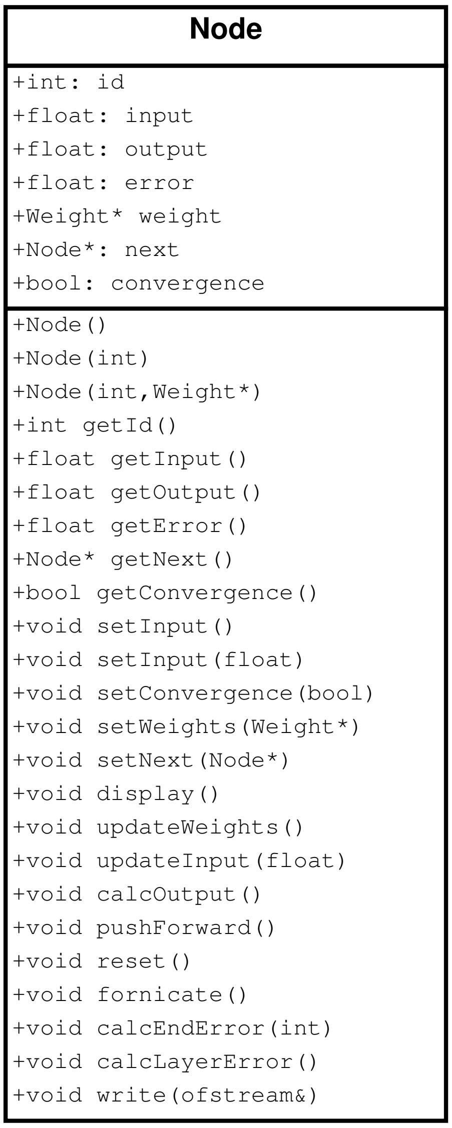
```python
char = "A"  # Initialized starting charachter "A"
targ = 1    # Initialized starting target for "A"
arr  = []   # Array to hold the 3x3 pixel subset, used to calculate average
count = 1;  # Initial count value

dData = open("data.dta", "w")           #Open file for writing testing data
for char in string.ascii_uppercase:     #Iterate through A..Z
  dData.write( char )
  tTxt = str(bin(targ))[2:]
  for i in range(26):
    if i == 26-targ:
      dData.write(" 1")
    else:
      dData.write(" 0" )
  dData.write("\n")

  for count in range(1, 21):            #For each letter go through 1 to 20
    iName = char + str(count)  + ".jpg" #Filename A1.jpg ... Z20.jpg
    print iName
    im = Image.open("pics/" + iName)    #Open the org 30x30 grayscale image
    imOrg = im.load()                   #LOad image to manipulate pixels
    imNew = Image.new("L", (30,30))     #Create new grayscale 10x10 image
    imNewP = imNew.load()               #Load image to manipulate pixels
    imgArr= []                          #Array to hold the 100 avg pixel values

    for y in range(0,10):     #Go through all rows in 10x10 image
      for x in range(0, 10):  #Go through all columns in 10x10 image
        arr = None            #Reset value array
        arr = [               #Create array from pixel values of the 3x3 subset
          imOrg[x*3  ,   y*3],   imOrg[x*3+1  , y*3],   imOrg[x*3+2  , y*3],
          imOrg[x*3  ,   y*3+1], imOrg[x*3+1  , y*3+1], imOrg[x*3+2  , y*3+1],
          imOrg[x*3  ,   y*3+2], imOrg[x*3+1  , y*3+2], imOrg[x*3+2  , y*3+2]
        ]
        imNewP[x*3,   y*3] = sum(arr)/len(arr)    #Insert avg pixel val to image
        imNewP[x*3+1, y*3] = sum(arr)/len(arr)    #Insert avg pixel val to image
        imNewP[x*3+2, y*3] = sum(arr)/len(arr)    #Insert avg pixel val to image

        imNewP[x*3,   y*3+1] = sum(arr)/len(arr)  #Insert avg pixel val to image
        imNewP[x*3+1, y*3+1] = sum(arr)/len(arr)  #Insert avg pixel val to image
        imNewP[x*3+2, y*3+1] = sum(arr)/len(arr)  #Insert avg pixel val to image

        imNewP[x*3,   y*3+2] = sum(arr)/len(arr)  #Insert avg pixel val to image
        imNewP[x*3+1, y*3+2] = sum(arr)/len(arr)  #Insert avg pixel val to image
        imNewP[x*3+2, y*3+2] = sum(arr)/len(arr)  #Insert avg pixel val to image

        imgArr.append( sum(arr)/float(len(arr)) ) #Insert avg pixel val to array
    imNew.save("con/" + iName)  #Save the new 10x10 picture to con/[filename]

    for i in imgArr:             #For all values un the image array
      dData.write( str( float(i) ) + " " )
    dData.write( "\n" )
  targ += 1
  dData.close()
  raw_input("Check the written file")
  dData = open( "data.dta", "a" )
dData.close()
```

A.2 Image verification script

Code A.2: Data file displayer

```
inp = open( "data2.dta" )          #Open data set file
for c in range(0, 546):            #for all images in file(546 lies)
  image = inp.readline()           #read new line
  if c % 21 != 0 :                 #Skip all first lines(informative lines)
    print c%21                     #Print current image set
    image = image.strip(" \t\r\n")#remove pre/post whitespace
    image = image.split(" ")       #split into array
    for i in range( len(image) ): # for all items in array
      if i % 10 == 0:              #split lines into ten symbols
        print ""
      if float(image[i]) > 175: #If pixel value is white
        print " ",                 #Print space
      else:                        #If pixel value is dark
        print "#",                 #Print a "#"-symbold
                                   #Pause printing
    raw_input("Click for next image")
```

B Images, figures and illustrations



```
Node
+int: id
+float: input
+float: output
+float: error
+Weight* weight
+Node*: next
+bool: convergence
+Node()
+Node(int)
+Node(int,Weight*)
+int getId()
+float getInput()
+float getOutput()
+float getError()
+Node* getNext()
+bool getConvergence()
+void setInput()
+void setInput(float)
+void setConvergence(bool)
+void setWeights(Weight*)
+void setNext(Node*)
+void display()
+void updateWeights()
+void updateInput(float)
+void calcOutput()
+void pushForward()
+void reset()
+void fornicate()
+void calcEndError(int)
+void calcLayerError()
+void write(ofstream&)
```

Figure 3: UML depiction of the perceptron

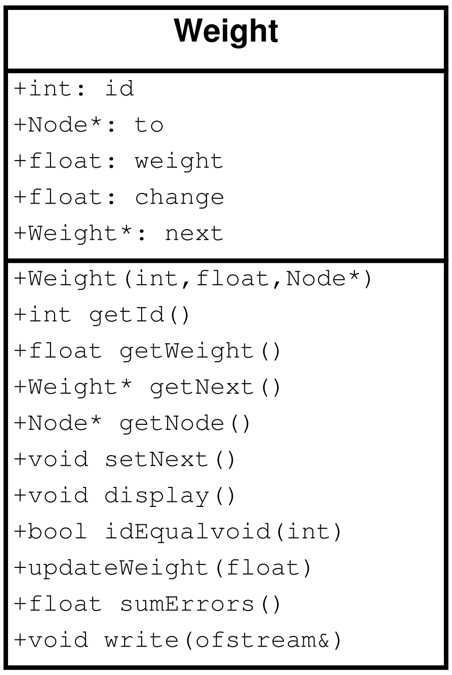


Figure 4: UML depiction of the synapses

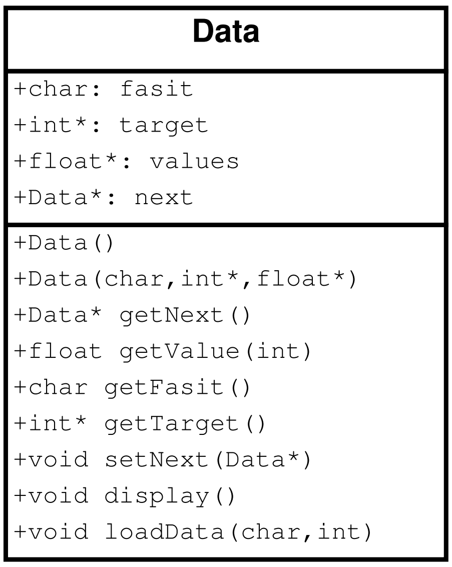


Figure 5: UML depiction of the image, data structure

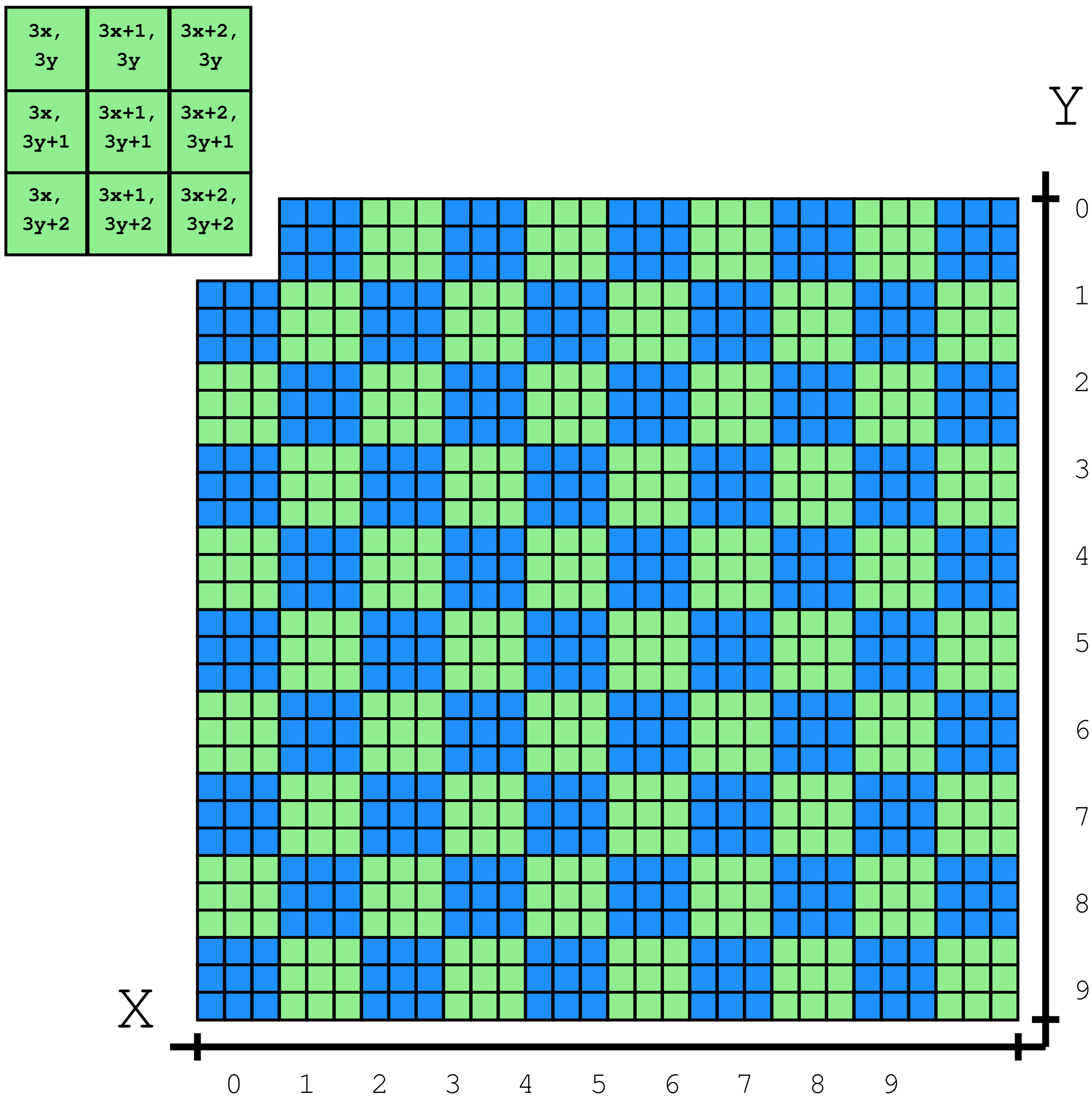


Figure 6: How the image is divided into subsets and scaled down