Obligatory Exercise 3

Group 8 - Report

090832	Magnus Øverbø
130197	Adrián Alberdi Ainziburu
131724	Miranda Qorolli

Contents

1   Implementation and running

1.1   Programs and Data Sets

1.1.1   Data generator

"create_data.py", this program uses the Pyton Image Library to read the images and output the data to a file called "data.dta". This file will in the end contain all the pixel values of the scaled down images. It will also save the scaled down images to a separate folder as a new image, just for fun.

1.1.2   Data verifier

This is a fun little Python script which reads the data file generated by the data generator. The program is the "checkData.py" executable. The program will read the data sets and project the "image" of the data to the user. The use of this program is to verify the data sets.

1.1.3   ANN trainer program

This program creates and trains the neural network by importing the data set file and using this data for training. The program to be run is the "main.cpp" which contains the C++ source code for the program. It will trian the network for x amount of times as specified by the source code. When the training is done it will save the ANN to a file, "savefile.net" which is then used by the testing program.

1.1.4   ANN testing program

The C++ file "eval.cpp" is the source code file for running the image data sets through a saved neural network. It will take the savefile generated from the previously run ANN trainer program. It will then run the set data sets and classify the output from the output layer.

1.1.5   Saved Files

Image Data Sets

"data2.dta" is the image data set file generated by "create_data.py". It is used by the data verifier, ANN trainer program and the ANN testing program.

ANN Save File

The files "save_0_10.net" and "save_11_20.net" is the saved files which was generated by the ANN trainer program and is used by the ANN testing program.

1.2   Running

Compile the program using Visual Studio on Windows or g++ on Linux. This wil compile the files and make an executable. When using Visual Studio on Windows make sure to correct all the file paths since it puts the output files to a different directory than the source code directory.

On linux you can run the make tool to compile the programs. "make learn" will create the ANN learning program, "make test" will create the ANN testing program.

2   Image Data

2.1   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.

2.2   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..

2.2.1   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.

2.3 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.

2.3.1 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.

3 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.

3.1 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.

3.1.1 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.
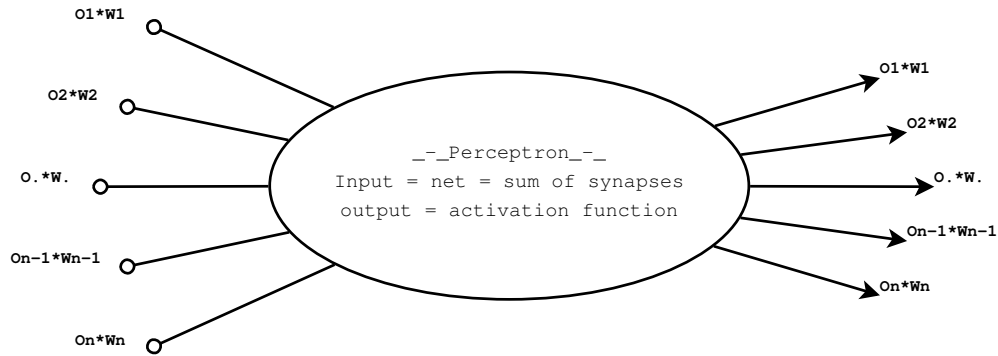

Figure 1: Illustration of a perceptron with n inputs and n outputs

3.1.2 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.

3.2 Synapses (Weights)

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 4.

3.2.1 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".

3.3 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.
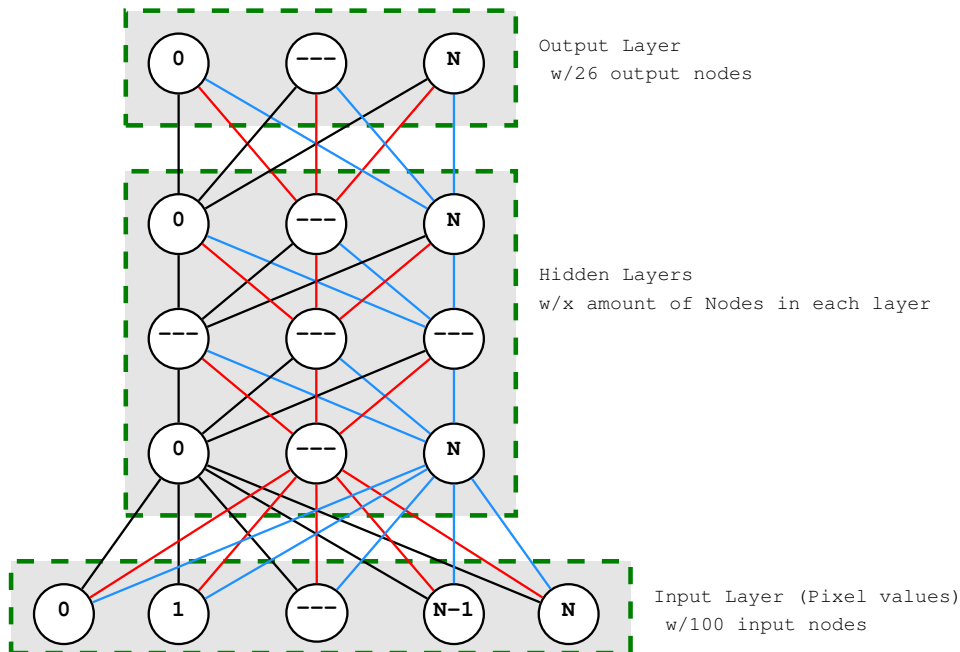
Figure 2: Illustration of a fully connected, neural network

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.

3.3.1 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.

3.3.2 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.

3.3.3 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%

4 Backpropagation Algorithm

4.1 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.

4.2 Errors

4.2.1 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.

4.2.2 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

4.2.3 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

4.3 Weight changes

4.3.1 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.

4.3.2 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.

5 Neural Network Training and Testing

5.1 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.

5.2 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.

6 Results

6.1 Learning set 1-10

6.1.1 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
A	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	91%
B	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%
C	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%
D	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	2%	0%	0%	0%	0%	100%	0%	0%	0%
E	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%
F	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%	0%	91%	0%	1%	0%	0%	0%
G	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	99%	0%	0%	0%	0%	0%	0%
H	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	9%	0%	0%	91%	0%	0%	0%	0%	0%	0%	0%
I	0%	0%	3%	0%	0%	0%	0%	9%	0%	0%	0%	0%	0%	0%	0%	0%	0%	92%	0%	0%	0%	0%	0%	0%	0%	0%
J	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
K	0%	0%	7%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	91%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
L	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
M	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
N	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	1%	91%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
O	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%
P	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Q	0%	0%	0%	0%	0%	0%	0%	0%	0%	99%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
R	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
S	0%	0%	0%	0%	0%	0%	0%	91%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	6%	0%	0%	0%	0%	0%	0%	0%
T	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
U	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
V	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
W	0%	0%	0%	91%	0%	9%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
X	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Y	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Z	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

Table 4: The performance data for AI trained with data set 1-10 and tested with data set 1-10

6.1.2 Set 11-20

#	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
A	0%	0%	34%	0%	0%	0%	0%	0%	0%	10%	0%	37%	24%	0%	0%	0%	43%	0%	0%	0%	0%	0%	0%	0%	0%	0%
B	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	2%	0%	68%	0%
C	0%	0%	0%	0%	0%	0%	29%	0%	0%	0%	2%	0%	0%	16%	0%	0%	0%	9%	0%	13%	19%	1%	0%	78%	0%	0%
D	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	69%	0%	0%	0%	69%	0%	10%	0%
E	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	92%	0%	1%	26%	1%
F	0%	0%	0%	0%	0%	0%	8%	0%	0%	0%	15%	0%	0%	0%	0%	0%	0%	0%	0%	0%	86%	0%	0%	0%	0%	0%

#	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
G	0%	0%	6%	0%	3%	1%	0%	0%	0%	1%	10%	18%	0%	0%	9%	0%	1%	0%	6%	43%	9%	0%	0%	2%	0%	2%
H	0%	0%	0%	0%	0%	31%	0%	0%	0%	3%	0%	35%	0%	0%	0%	10%	0%	0%	57%	0%	0%	0%	2%	0%	0%	0%
I	3%	1%	6%	10%	0%	0%	0%	0%	0%	0%	0%	8%	0%	0%	0%	0%	1%	86%	0%	0%	0%	0%	0%	0%	0%	0%
J	12%	4%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	10%	0%	92%	0%	0%	0%	0%	0%	14%	0%	0%	0%
K	0%	0%	0%	0%	0%	0%	9%	0%	0%	0%	0%	0%	2%	70%	0%	93%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
L	0%	0%	0%	0%	0%	58%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	5%	0%	1%	0%	0%	0%	0%
M	0%	0%	0%	0%	0%	3%	10%	0%	1%	0%	0%	0%	0%	11%	0%	0%	0%	0%	3%	0%	0%	0%	46%	0%	10%	0%
N	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	5%	74%	0%	0%	0%	0%	0%	20%	0%	3%	0%	0%	0%	0%	0%
O	12%	0%	0%	0%	3%	2%	0%	0%	0%	0%	4%	96%	0%	0%	6%	0%	29%	8%	0%	4%	0%	0%	0%	0%	0%	0%
P	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	96%	0%	0%	0%	0%	0%	0%	0%	0%	3%	0%	8%	0%	0%	0%	0%
Q	6%	0%	0%	0%	0%	0%	0%	3%	0%	56%	1%	14%	0%	0%	0%	0%	0%	0%	16%	3%	13%	0%	0%	0%	0%	0%
R	0%	0%	0%	0%	0%	0%	10%	0%	82%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
S	2%	0%	9%	0%	0%	0%	0%	34%	0%	0%	2%	0%	0%	0%	0%	0%	0%	17%	0%	10%	0%	6%	0%	16%	9%	0%
T	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%
U	0%	0%	0%	21%	0%	79%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
V	0%	39%	0%	0%	60%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
W	0%	0%	0%	60%	0%	27%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%
X	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%
Y	0%	91%	0%	0%	0%	0%	10%	0%	2%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%
Z	90%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	10%	0%	0%	0%

Table 5: The performance data for AI trained with data set 1-10 and tested with data set 11-20

6.2 Learning Set 11-20

6.2.1 Set 1-10

#	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
A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	90%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	9%
B	8%	0%	0%	0%	0%	0%	0%	20%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	1%	0%	44%	0%	0%
C	0%	0%	0%	0%	0%	0%	1%	0%	0%	1%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	1%	74%	9%	0%	0%
D	0%	0%	0%	0%	0%	0%	0%	0%	0%	4%	0%	0%	0%	9%	0%	0%	0%	0%	0%	0%	0%	72%	9%	9%	0%	0%
E	0%	0%	0%	0%	0%	0%	18%	9%	0%	0%	0%	0%	0%	0%	0%	10%	0%	0%	0%	0%	0%	54%	0%	0%	0%	0%
F	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	2%	0%	0%	0%	3%	95%	5%	0%	0%	0%	4%
G	0%	0%	0%	0%	0%	0%	0%	0%	0%	18%	0%	0%	0%	0%	0%	0%	0%	0%	0%	75%	0%	0%	0%	1%	0%	0%
H	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	1%	0%	11%	0%	0%	0%	0%	0%	85%	0%	0%	0%	0%	0%	0%	0%
I	0%	0%	0%	9%	0%	0%	0%	1%	0%	0%	0%	0%	7%	0%	0%	0%	0%	51%	0%	1%	1%	4%	0%	7%	0%	7%
J	3%	0%	0%	0%	22%	26%	0%	18%	13%	2%	0%	0%	0%	1%	0%	0%	62%	0%	0%	0%	0%	0%	1%	0%	31%	0%
K	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	4%	0%	8%	82%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
L	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	74%	0%	27%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%
M	0%	0%	0%	0%	0%	0%	0%	0%	9%	0%	0%	9%	25%	55%	1%	0%	0%	0%	0%	0%	0%	0%	9%	0%	0%	0%
N	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	72%	19%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
O	0%	0%	0%	0%	0%	2%	0%	0%	0%	26%	0%	72%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	6%	0%	0%	0%
P	0%	0%	0%	6%	0%	0%	0%	0%	0%	0%	68%	0%	0%	0%	0%	0%	0%	2%	2%	0%	18%	0%	0%	0%	0%	8%
Q	0%	0%	0%	0%	0%	0%	0%	1%	0%	46%	0%	0%	0%	7%	0%	0%	0%	0%	0%	0%	0%	19%	8%	17%	0%	0%
R	0%	0%	0%	0%	0%	0%	0%	0%	99%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
S	0%	0%	0%	0%	0%	0%	0%	85%	0%	11%	0%	0%	0%	12%	0%	0%	0%	0%	0%	0%	0%	1%	0%	11%	0%	0%
T	5%	0%	0%	0%	0%	0%	77%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	8%	0%	0%	0%	0%
U	0%	0%	10%	0%	31%	83%	0%	0%	0%	0%	0%	0%	0%	0%	13%	0%	0%	0%	1%	0%	0%	0%	8%	0%	1%	0%
V	9%	0%	1%	23%	97%	11%	0%	24%	0%	0%	0%	0%	0%	0%	0%	0%	0%	5%	0%	0%	0%	0%	0%	0%	0%	0%

#	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
W	0%	0%	0%	82%	0%	11%	0%	0%	0%	0%	0%	0%	0%	9%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
X	0%	8%	87%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Y	6%	27%	4%	9%	46%	0%	0%	9%	9%	0%	0%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	0%
Z	69%	1%	0%	0%	0%	0%	0%	0%	0%	0%	19%	0%	0%	0%	0%	0%	0%	1%	0%	0%	0%	0%	0%	0%	0%	19%

Table 6: The performance data for AI trained with data set 11-20 and tested with data set 1-10

6.2.2 Set 11-20 - 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
A	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%
B	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%
C	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%
D	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%
E	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%
F	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%
G	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%
H	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%
I	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	90%	0%	0%	0%	0%	0%	0%	0%	0%
J	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%
K	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
L	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
M	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
N	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
O	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
P	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Q	0%	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
R	0%	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
S	0%	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
T	0%	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
U	0%	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
V	0%	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
W	0%	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
X	0%	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Y	0%	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%
Z	100%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%	0%

Table 7: The performance data for AI trained with data set 11-20 and tested with data set 11-20

7 Performance

7.1 Learning

During our learning sessions we did try different values for our network, but didn't try different topologies after doing the initial tests. We landed upon having a topology with a single hidden layer consisting of 10 perceptrons. The learning algorithm with back propagation was then run a 100.000 times on the entire testing dataset to reduce the errors. After a couple of initial tests at 30.000 runs, there was still some perceptrons that didn't converge fully but had an offset from the target value of 0.01.

This was by far the worst offset from target value, but mostly because we had set the threshold at 0.005.

There are also some other choices we made which may have resulted in a worse time performance since we are allways doing weight updates, even when the perceptron is within its target output. But we have not made these tests to know for sure.

Below we have recorded some statistics of how long it took to learn and run the different set, to illustrate the differences. As you can see the last run had a maximum offset from target of 0.0059, which is more than tolerable and adequate for testing. The calculated error values for the perceptron is considered equal to zero since they are so low.

Data set	Rounds	Time		Error	Offset Target
1-10	100.000	00:24:54.194	MAX	0	0.0059
			MIN	0	-0.0038
			AVG	0	-0.0001
11-20	100.000	00:25:15.142	MAX	0	0.0045
			MIN	0	-0.0035
			AVG	0	-0.0001

7.2 Testing

For our testing we ran four test using our final 2 sets of network training weights. We ran, as show in the next chapter, the four tests: learn data set 1-10 and test both data set 1-10 and 11-20 against the resulting network. Learn data set 11-20 and test both data set 1-10 and 11-20 against the resulting network. The resulting information is then feed into a spreadsheet program as a CSV file where we do some quick calculations and editing before exporting them to the report format shown in the result chapter.

The summary of the data presented in the next chapter are the averaged values of the the 10 data sets it is being ran through. This means that the represented values in the tables are the average of 10 runs per character. This means that some of the data sets will have a different value, than the rest.

A short summary of the data is that we have a good network that can identify very well the

data set it was trained with. It performs adequately against the opposite dataset in accordance to which we trained with. When testing the opposite data set we receive a good amount of errors, but it is mostly able to identify the correct characters with a mid-range certainty.

Overall performance of the network is relatively good. It should be pointed out that we could have implemented more tests and tested different topologies, but due to programming errors and wasted time it was left undone. But the program is very modular and can easily be used for testing.

7.2.1   Data set review

As we can see from the two tables, table 4 and 7, where we have trained and tested using the same data sets. The reckognition rate for the characters are between 90% and 100%. which is good results but very biased since this is the data we have trained with.

As is clearly shown in table 4 and 7 you can see that the resulting matrix has a diagonal. This diagonal indicates that the correct character was identified with a very good rate.

If we instead look at the tests where we ran the data set which we didn't train with. We can here see that the reckognition of characters is not as good as the previous test. The values for each character is more spread than the "same-set" tests. Even though the reckognition rates for the characters are being spread out, it still has the reckognizable diagonal which identifies the letters as the correct character with a significant higher value than the rest of the values.

As is clearly shown in table 5 and 6 you can see that the resulting matrix has a diagonal. This diagonal indicates that the correct character was identified with a very good rate.

List of Figures

Code snippets and scripts

List of Tables

List of Abbreviations

AI   Artificial Intelligence

ANN   Artificial Neural Network

GUC   Gjøvik University College

HiG   Høgskolen i Gjøvik

PIL   Python Image Library

Python   A scripting language with a ton of usefull functionality

UML   Unified Modeling Language

Glossary

activation function A function inside the perceptron that calculates a desired output value. 4

feed forward A network that only pushes values from the front to the end. 5

neural network A network of perceptrons connected in a network with the use of synapses 4

perceptron A node that receives a set of weighted, input values and calculates a single output value. 3–5, 7

Sigmoid A function that generates an S-shaped graph. 4

synapse A one way connector between two perceptron, that provides a weighted value as input to the ending perceptron 4, 5

A   Scripts and Code Snippets

A.1   Image Converter

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

```
1  char = "A"   # Initialized starting charachter "A"
2  targ = 1     # Initialized starting target for "A"
3  arr  = []    # Array to hold the 3x3 pixel subset, used to calculate average
4  count = 1;   # Initial count value
5
6  dData = open("data.dta", "w")         #Open file for writing testing data
7  for char in string.ascii_uppercase:   #Iterate through A..Z
8    dData.write( char )
9    tTxt = str(bin(targ))[2:]
10   for i in range(26):
11     if i == 26-targ:
12       dData.write(" 1")
13     else:
14       dData.write(" 0" )
15   dData.write("\n")
16
17   for count in range(1, 21):            #For each letter go through 1 to 20
18     iName = char + str(count)  + ".jpg" #Filename A1.jpg ... Z20.jpg
19     print iName
20     im = Image.open("pics/" + iName)    #Open the org 30x30 grayscale image
21     imOrg = im.load()                     #LOad image to manipulate pixels
22     imNew = Image.new("L", (30,30))     #Create new grayscale 10x10 image
23     imNewP = imNew.load()                 #Load image to manipulate pixels
24     imgArr= []                            #Array to hold the 100 avg pixel values
25
26     for y in range(0,10):      #Go through all rows in 10x10 image
27       for x in range(0, 10):   #Go through all columns in 10x10 image
28         arr = None               #Reset value array
29         arr = [                  #Create array from pixel values of the 3x3 subset
30           imOrg[x*3 ,   y*3],   imOrg[x*3+1 , y*3],   imOrg[x*3+2 , y*3],
31           imOrg[x*3 ,   y*3+1], imOrg[x*3+1 , y*3+1], imOrg[x*3+2 , y*3+1],
32           imOrg[x*3 ,   y*3+2], imOrg[x*3+1 , y*3+2], imOrg[x*3+2 , y*3+2]
33         ]
34         imNewP[x*3,   y*3] = sum(arr)/len(arr)     #Insert avg pixel val to image
35         imNewP[x*3+1, y*3] = sum(arr)/len(arr)     #Insert avg pixel val to image
36         imNewP[x*3+2, y*3] = sum(arr)/len(arr)     #Insert avg pixel val to image
37
38         imNewP[x*3,   y*3+1] = sum(arr)/len(arr)  #Insert avg pixel val to image
39         imNewP[x*3+1, y*3+1] = sum(arr)/len(arr)  #Insert avg pixel val to image
40         imNewP[x*3+2, y*3+1] = sum(arr)/len(arr)  #Insert avg pixel val to image
41
42         imNewP[x*3,   y*3+2] = sum(arr)/len(arr)  #Insert avg pixel val to image
43         imNewP[x*3+1, y*3+2] = sum(arr)/len(arr)  #Insert avg pixel val to image
44         imNewP[x*3+2, y*3+2] = sum(arr)/len(arr)  #Insert avg pixel val to image
45
46         imgArr.append( sum(arr)/float(len(arr)) ) #Insert avg pixel val to array
47     imNew.save("con/" + iName)  #Save the new 10x10 picture to con/[filename]
48
49     for i in imgArr:                #For all values un the image array
50       dData.write( str( float(i) ) + " " )
```

```
51       dData.write( "\n" )
52     targ += 1
53     dData.close()
54     raw_input("Check the written file")
55     dData = open( "data.dta", "a" )
56 dData.close()
```

A.2   Image verification script

Code A.2: Data file displayer

```
1 inp = open( "data2.dta" )            #Open data set file
2 for c in range(0, 546):             #for all images in file(546 lies)
3   image = inp.readline()            #read new line
4   if c % 21 != 0 :                  #Skip all first lines(informative lines)
5     print c%21                      #Print current image set
6     image = image.strip(" \t\r\n")#remove pre/post whitespace
7     image = image.split(" ")        #split into array
8     for i in range( len(image) ): # for all items in array
9       if i % 10 == 0:              #split lines into ten symbols
10        print ""
11      if float(image[i]) > 200: #If pixel value is white
12        print " ",                  #Print space
13      else:                         #If pixel value is dark
14        print "#",                  #Print a "#"-symbold
15                                     #Pause printing
16    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

```
┌─────────────────────────────────────┐
│              Weight                 │
├─────────────────────────────────────┤
│ +int: id                            │
│ +Node*: to                          │
│ +float: weight                      │
│ +float: change                      │
│ +Weight*: next                      │
├─────────────────────────────────────┤
│ +Weight(int,float,Node*)            │
│ +int getId()                        │
│ +float getWeight()                  │
│ +Weight* getNext()                  │
│ +Node* getNode()                    │
│ +void setNext()                     │
│ +void display()                     │
│ +bool idEqualvoid(int)              │
│ +updateWeight(float)                │
│ +float sumErrors()                  │
│ +void write(ofstream&)              │
└─────────────────────────────────────┘
```

Figure 4: UML depiction of the synapses

```
┌─────────────────────────────────────┐
│               Data                  │
├─────────────────────────────────────┤
│ +char: fasit                        │
│ +int*: target                       │
│ +float*: values                     │
│ +Data*: next                        │
├─────────────────────────────────────┤
│ +Data()                             │
│ +Data(char,int*,float*)             │
│ +Data* getNext()                    │
│ +float getValue(int)                │
│ +char getFasit()                    │
│ +int* getTarget()                   │
│ +void setNext(Data*)                │
│ +void display()                     │
│ +void loadData(char,int)            │
└─────────────────────────────────────┘
```
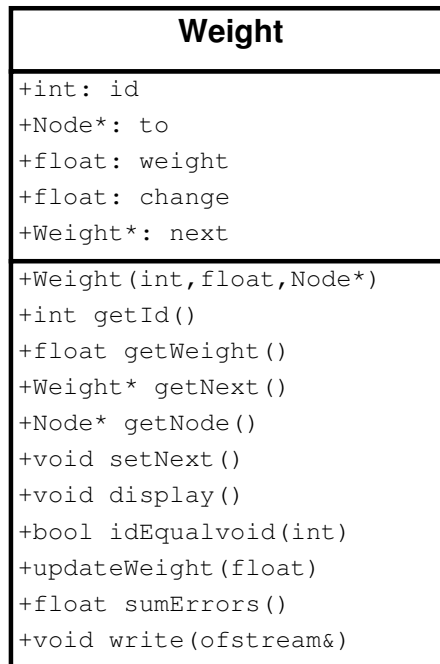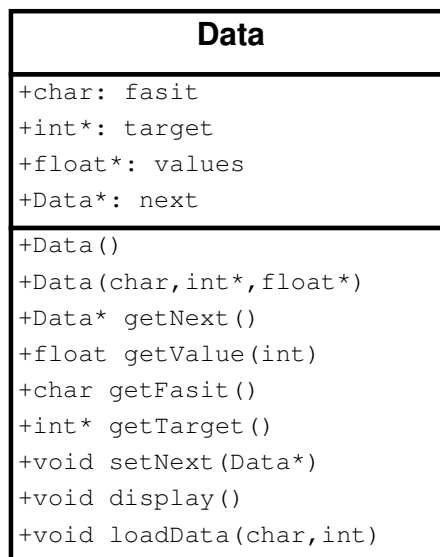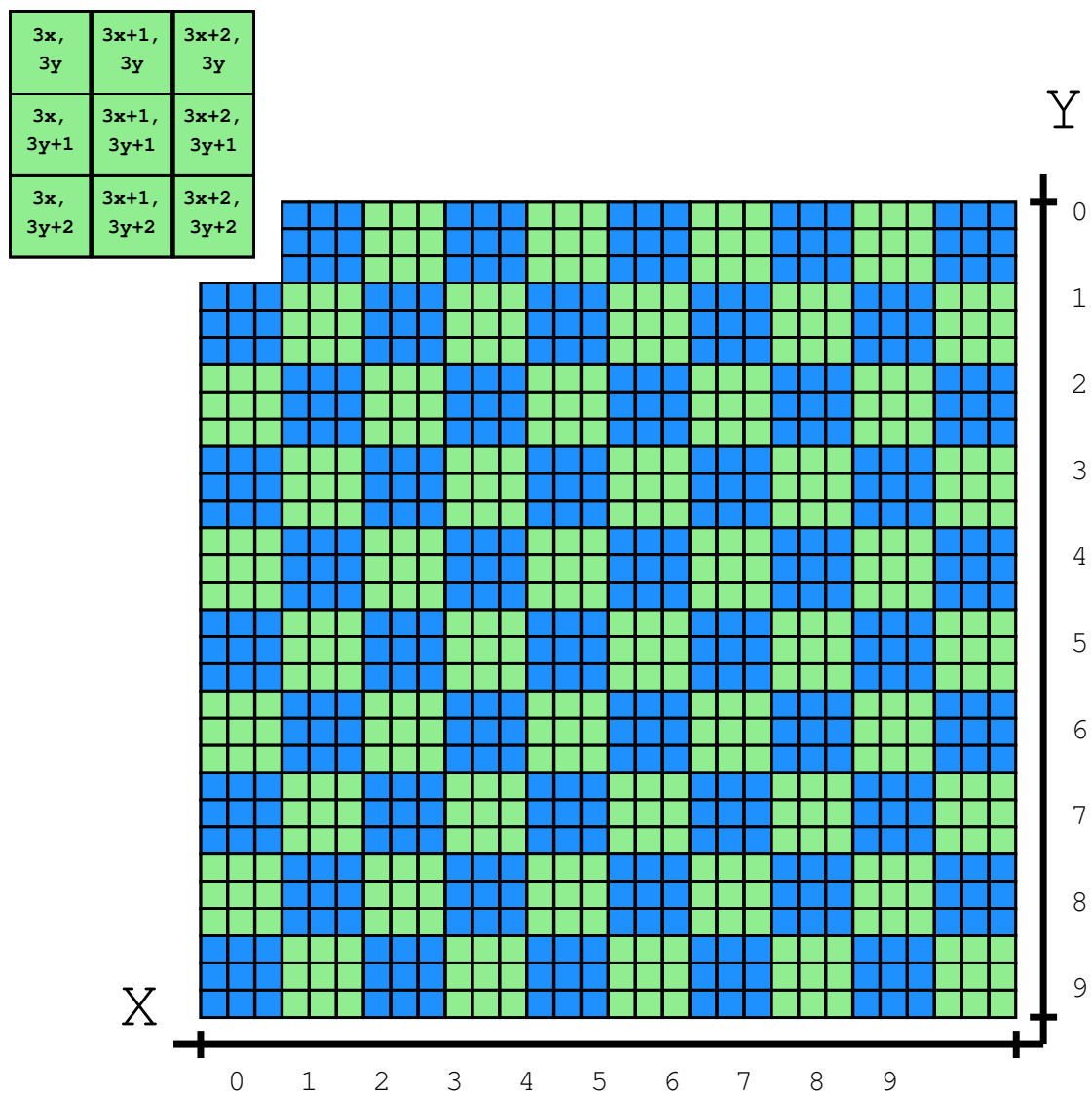
Figure 5: UML depiction of the image, data structure

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