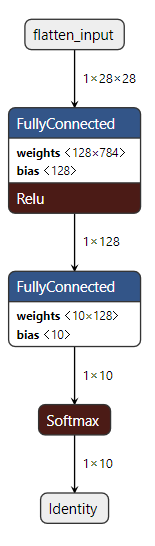
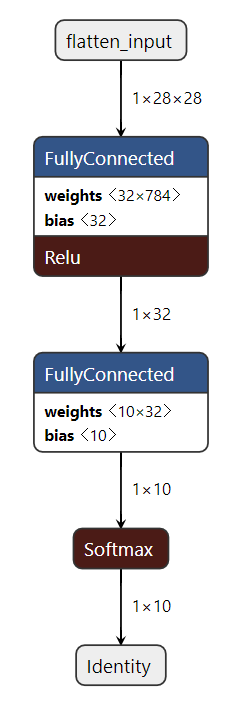
## Creating a neural network

As the classification problem is a multi class problem with 10 classes, a network must be designed in to accommodate this. The model will also be designed to run on a embedded processor with limited space and computational power. If possible the model should also avoid floating point arithmetic as this will impact the model in a negative way. As this process is a interactively task the google COLAB framework is used in collaboration with KERAS. 3 different nets is developed in order to make some design space exploration.

### Simple neural network with no data processing

To get a baseline model for the problem a simple neural network is developed. This network does minimum processing on the input data and simply uses one hidden layer and one output layer. The input in this model is the full 28x28 image and a hidden layer with 128 neurons gives a network as seen in figure \ref{fig:SimpleNetwork}. Here it can be concluded that the model uses 103512 weights to complete the task and gets a precision around 93.7% on the validation set.

### Neural network with Harris Corner detection

To lower the amount of weights in the network some prepossessing of the image data is used. Here a Harris Corner detector{ref to Harris corner detector} is used to augment the input images. The input size of the model is still 28X28 but a hidden layer with 28 neurons is used giving the model a smaller number of weights. The network can be seen in figure \ref{fig:CornerNetwork}. This model uses 27832 weights while having a precision of 89.3%. But this network also uses some processing time on detection corners in the picture. A more optimized method of only giving the network a fixed amount of corners will lower the input size of the network.

### Neural network with gaussian down sampling

To get the number of weights further down in the network. Some image processing must be applied to the image in order to down sample the input size. Through experiments it is found that the original input image with a size of 28 times 28 pixels can be down sampled with a gaussian pyramid to a input size of 4 times 4. This means it is hard for a human to classify the images. But the network performs with a accuracy of around 86%. The network is now so small that it can be drawn in figure {REF}. This model only have 4732 weights in total meaning that it can run on very small embedded platforms.

## Converting the best model to uint8 format

As the neural network with gaussian down sampling is by far the most suitable network for a embedded system this model is converted from using floating point weights to use unsinged 8 bit integers. This means the underlying matrix multiplications can be done efficiently on every platform. As the TensorFlow library can convert a model with a few simple lines of code, this is a trivial task.

# Implementing the neural network in C

To make the implementation possible on a embedded platform a simply implantation is created in C. This is done to ensure the correct design methods is used. The OpenCV library is used to handle image files directly from the computers storage. When the image is loaded into the programs memory the implementation is the same as the software running on the embedded platform. The mathematics are simple. Each neuron in the hidden layer hides 2 different purposes. Firstly calculation the weighted average of the input and secondly parse the weighted average through a activation function. The first part can be described with the following:

Here **W** is the weight vector and b is the bias. Both are obtained through training the network. The training is already done and these values are given to the program. The second part is parsing the result of this calculation though a non linear activation function. This is done with g being the function:

To keep the model simple the activation function for this model is a rectified linear unit often referred to as a RELU function. The function can be described as the following:  
meaning that every negative number will be zero.

## Implementing the down sampling in C

The following function is used in order to gaussian down sample the image. The image must be down sampled 3 times in order to get the image in a 4 by 4 size:

|  |
| --- |
| void pyDownsampleTo4x4(Mat\* img)  {  Mat\* tempResult = new Mat();  pyrDown(\*img,\*tempResult);  pyrDown(\*tempResult, \*img);  pyrDown(\*img, \*tempResult);  \*img = \*tempResult;  delete tempResult;  } |

The images is still in a 4 times 4 format at this point. And as the model only works on vectors the image is flatten:

|  |
| --- |
| void flatten(Mat\* img)  {  \*img = img->reshape(1, 1);  } |

The image is now in a 1 times 16 format.

## Developing a matrix and bias multiplication algorithm in software

As the image changes format throughout the model 2 functions is made to handle the multiplication. These functions must take a input image and return a single integer as result. Here the function is shown with a input of a normal array. The weights and the bias is saved static in the memory as these never change:

|  |
| --- |
| int matrixMultiplicationAndBias(int64\_t input[], const int8\_t W\_in[], const int8\_t bias, int size)  {  int i, result = 0;  for (i = 0; i < size; i++)  {  result += W\_in[i] \* input[i];  }  result += bias;  return result;  } |

## Implementing activations functions

As seen in the sketch of the model 2 activation functions is used. The first is a simple RELU function and is implemented with a if statement:

|  |
| --- |
| void RELU(int64\_t data[32])  {  for (int i = 0; i < 32; i++)  {  if (data[i] < 0)  {  data[i] = 0;  }  }  } |

Here it can be seen how the function keeps the positive numbers and make all negative numbers 0. The other activation function is softmax function. This function tells how certain the model is on its guess. The function can be described as the following:  
This is implemented with the following function:

|  |
| --- |
| int SoftMaxLayer(int64\_t input[])  {  int64\_t output[10];  for (int i = 0; i < 10; i++)  {  output[i] = matrixMultiplicationAndBias(input, W\_sm[i], b\_sm[i], 32);  }  int max = output[0];  int k = 0;  for (int i = 0; i < 10; ++i)  {  if (output[i] > max)  {  max = output[i];  k = i;  }  }  return k;  } |

## Tieng all implementations together

To make the hidden layer a function that calls the multiplication 32 times is created, it also call the activation function:

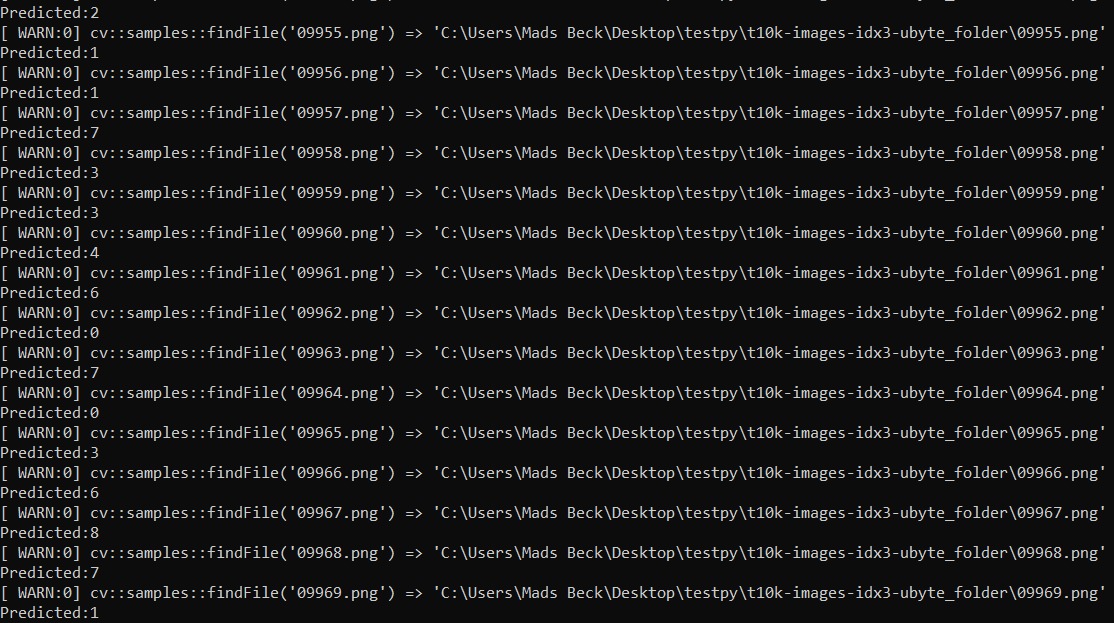
|  |
| --- |
| void NNLayer(Mat\* input,int64\_t output[32])  {  for (int i = 0; i < 32; i++)  {  output[i] = matrixMultiplicationAndBias(input, W[i], B[i],16);  }  RELU(output);  } |

Finally all the software fucntion is created to make the desired network. The primary fucntion is called in the following manner:

|  |
| --- |
| pyDownsampleTo4x4(&image);  flatten(&image);  image.convertTo(image, CV\_8S);  NNLayer(&image,out);  clas = SoftMaxLayer(out); |

Some code is created to write the class to a file and write the output to the screen.

## Running and testing the code and the model

The code is executed and the following results is given:  
  
It can be seen how the model finds the image and makes a prediction. A simple Matlab script checks the output of the model predictions against the actual classes. This gives a precession on 82.92%.

