# Multi-Layer Perceptrons for Detecting Handwritten Digits and Characters in C

December 16, 2019

# Contents

## 0.1 Foreword

When we decided to tackle this project, we weren't sure how far we could or even should go. Both of us have an intense passion for Artificial Intelligence and Machine Learning, which was further deepened by our experiences in our AI course this semester, as well as a better understanding of C and how we could arrive at the dream we weren't able to reach in Twixt last year: a self-adjusting or "learning" system. Our appreciation of mathematical principles and progress in reading from and storing information in files also aided us in reaching this goal.

In deciding to treat and analyze images, we studied and applied different filters to get ideas about where to go beyond simple edge and shape detection based on conditions. We decided to give ourselves a greater challenge: to detect and recognize digits in an image, and perhaps even letters.

In this report we will show a few key functions and structures from our program, and illustrate the general processes and workings involved.

## 0.2 Overview

We have a couple of things to examine here: Perceptrons, our formatting of images, our analysis of images, and the EMNIST Database from the National Institute of Standards and Technology (NIST).
We will begin with a description of our Perceptron neuron.

## 0.3 The Perceptron

The Perceptron is an example of a supervised learning algorithm applied to a binary classifier. This means that we have examples of input data to which can compare our classification predictions and see whether they are correct or not. If the data are linearly separable in the dimension the Perceptron is working in, the Perceptron can find a hyperplane decision boundary to separate two classes, thus enabling prediction for data of unknown classification.

These decision boundaries are represented by weights which correspond to each of the input data's features, plus one bias weight. Our images are 28x28 pixels, so that is a feature input of 784 + 1. These weights are initialized and modified during the training phase (where we use our knowledge of the correct classification in order to move our decision boundary) using the Perceptron algorithm.

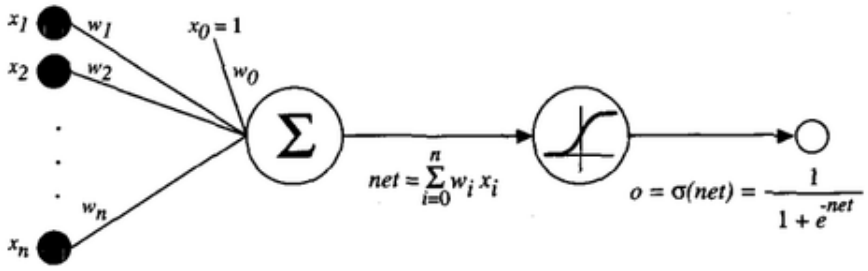
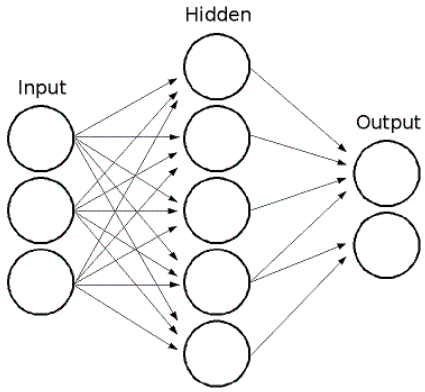



Figure 1: Sigmoid-Activated Perceptron

Our Perceptron uses a sigmoid activation function, which places its outputs between 0 and 1. During the training phase, we feed images to our neuron knowing how they are supposed to be classed. The difference between the target value and the output value is then propagated to the weights in the neuron that are associated either to each neuron of the previous layer, or in our case, each feature of the data, plus a bias value. This makes 785 weights in our Perceptron that change according to their proportion of responsibility for the error.




Multi-Layer Perceptron (MLP)

Ideally, we would have a Multi-Layered Perceptron (MLP). The basic idea behind this is that the error from the output is spread backward, or backpropagated, to the neurons in previous layers based on proportions of weight andand output, thus modifying their weights in order to minimize the total error of the output.

## 0.4 The Perceptron Algorithm

### 0.4.1 Forward Propagation

$\text{Output}_{total} = \sum(inputs * weights)$

```
void layer_output(matrix * previous_layer_output, layer * lyr){

  int i,j;
\newpage

  matrix * preactivation_matrix;
  matrix obj1;
  preactivation_matrix = &obj1;
  preactivation_matrix = matrix_dot_product(previous_layer_output,
                                    lyr->weights, preactivation_matrix);
  matrix_sigmoid(preactivation_matrix);
  for(i=0; i<preactivation_matrix->mat_w; i++){
    for(j=0; j<preactivation_matrix->mat_h; j++){
      lyr->outputs->mat[j][i] = preactivation_matrix->mat[j][i];
    }
  }
}

void forward(main_network * main_ntw, int indice_network){
  int i;

  matrix ** pt_data;
  pt_data = malloc(sizeof(matrix*));
  pt_data=&(main_ntw->current_batch_data);

  for(i=0;i<main_ntw->ntw[indice_network].nb_layer;i++){

    layer_output(*pt_data,&(main_ntw->ntw[indice_network].lyr[i]));
    pt_data=&(main_ntw->ntw[indice_network].lyr[i].outputs);
  }
}
```

### 0.4.2 Calculating Error

$\text{Error}_{total} = \sum(output - target)$

```
void whatdiduexpected(matrix * current_batch_class, int expected,
                                                    matrix * matrix_expected){
  int i;

  for(i=0;i<matrix_expected->mat_h;i++){
    if(((int)current_batch_class->mat[i][0])==expected){
      matrix_expected->mat[i][0]=1;
    }
    else{
      matrix_expected->mat[i][0]=0;
    }
  }
}

void total_error(main_network * main_ntw, layer * last_lyr,
                                                  int nrn_class){
  int i;
  float error;
  float negative_target;
  error=0;
  negative_target=0;

  matrix * matrix_expected;
  matrix_expected = malloc(sizeof(matrix));
  matrix_expected->mat_h = main_ntw->batch_size;
  matrix_expected->mat_w = 1;
  alloc_matrix(matrix_expected);
  whatdiduexpected(main_ntw->current_batch_class, nrn_class, matrix_expected);

  for(i=0;i<main_ntw->batch_size;i++){
    error += last_lyr->outputs->mat[i][0] - matrix_expected->mat[i][0];

    negative_target +=
            ((matrix_expected->mat[i][0])-(last_lyr->outputs->mat[i][0]));
  }

  last_lyr->error->mat[0][0] = error;
  last_lyr->negative_target = negative_target;
  free(matrix_expected);
}
```

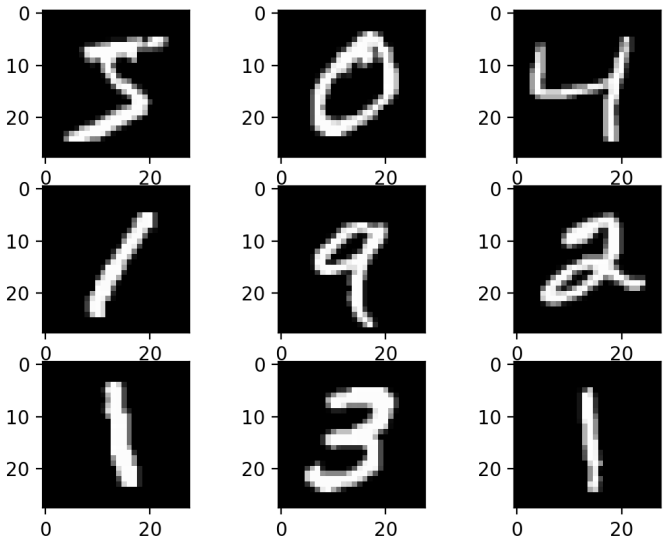
### 0.4.3 Backpropagation and Calculating New Weights

$w^+ = w - \eta * \frac{\partial E_{total}}{\partial w}$

```
void new_final_layer_weights(main_network * main_ntw,
            layer * last_lyr, layer * second_to_last, int nb_layer){

  int i, j;
  float delta;

  for(i=0; i<last_lyr->weights->mat_h; i++){
    delta = 0;

    if(nb_layer>1){
      for(j=0;j<main_ntw->batch_size;j++){
            delta += last_lyr->outputs->mat[j][0] *
                ( 1 - last_lyr->outputs->mat[j][0]) *
                    second_to_last->outputs->mat[j][i];
      }
    } else {
      for(j=0;j<main_ntw->batch_size;j++){
            delta += last_lyr->outputs->mat[j][0] *
                ( 1 - last_lyr->outputs->mat[j][0]) *
                    main_ntw->current_batch_data->mat[j][i];
      }
    }
    last_lyr->weights->mat[i][0] -=
        STEPSIZE * delta * last_lyr->error->mat[0][0];
  }
}

void new_final_layer_weights(main_network * main_ntw,
    layer * last_lyr, layer * second_to_last, int nb_layer){

  int i, j;
  float delta;

  for(i=0; i<last_lyr->weights->mat_h; i++){

    delta = 0;

    if(nb_layer>1){
      for(j=0;j<main_ntw->batch_size;j++){
        delta += last_lyr->outputs->mat[j][0] *
```

```
            ( 1 - last_lyr->outputs->mat[j][0]) *
            (second_to_last->outputs->mat[j][i]) /
                                        main_ntw->batch_size;
        }
      }

      else{
        for(j=0;j<main_ntw->batch_size;j++){
          delta += last_lyr->outputs->mat[j][0] *
            ( 1 - last_lyr->outputs->mat[j][0]) *
            (main_ntw->current_batch_data->mat[j][i]) /
                                        main_ntw->batch_size;
        }
      }

      last_lyr->weights->mat[i][0] -=
          STEPSIZE  *100 * delta * last_lyr->error->mat[0][0];
    }
  }
```

## 0.5 Formatting Images and the EMNIST Database



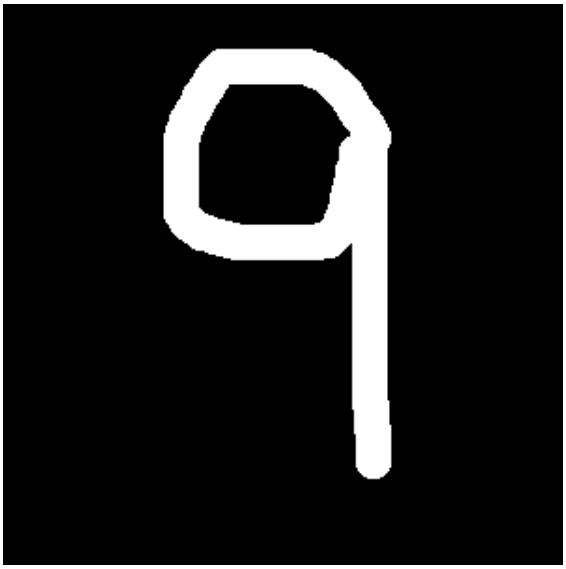
EMNIST Handwritten Digit Samples

In order to be able to read a digit or character in an image, we came up with the solution of identifying and separating single digits and characters based on their surrounding white space. Once identified, this subsection of image is then converted to a 28x28 image using mean pixel value, with the character or digit centered within the approximately 26x26 pixel center of the image. In order to successfully predict a digit or character, we have gone with white on a black background, which corresponds to the format of the EMNIST database. We found that the best results for hand-written digit recognition were for images with proportions similar to those of EMNIST. 500x500 pixels with a number drawn in 65-70 pixel thickness as a white-on-black digit worked quite well, for example.

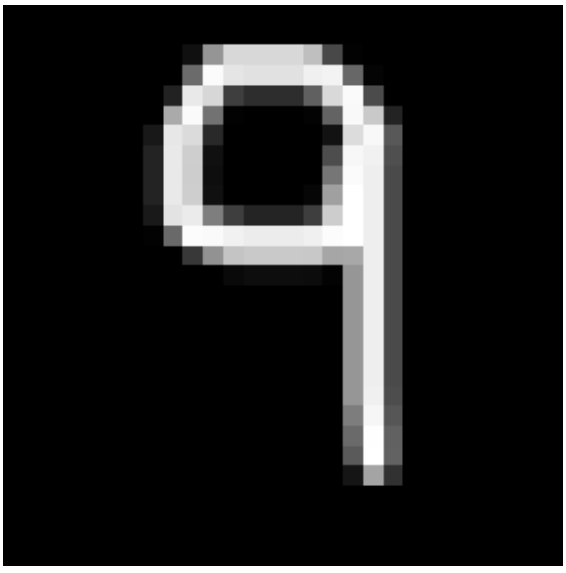
The EMNIST digits are provided with each image as a 785-pixel array; the first element of the array being its classification and the rest a number between 0 and 255 representing black to white.

On the following page is an example of a hand-drawn digit (using GIMP) that we convert to a 28x28 pixel image in order to conform with EMNIST. The processed version formatted for analysis by our Perceptron can be seen below

it.



Example of Hand-Drawn Digit Using GIMP



Example of Hand-Drawn Digit Processed and Formatted for Analysis

This was achieved by first identifying a digit's dimensions and placement in a

larger image, then taking mean gray values for blocks of pixels in order to resize the digit to the appropriate 26x26 pixels on a 28x28 background.

## 0.6 Image Processing

**Here we have some examples of the image processing library that we wrote:**

### 0.6.1 Reading the Image

```
image * lireExtraire(image * pic, char * nom){
  int val;
  val=0;

  int ix, iy;
  FILE * fd;
  char n;
  char * info;
  unsigned char a;

  info = malloc (sizeof(*info));
  assert(info);
  *info='\0';
  fd=ouvrir(nom);
  fscanf(fd,"%c",&n);
  fscanf(fd,"%c",&n);
  info=ajoutChar(info,n);
  pic->entete=atoi(info);
  resetStr(info);
  fscanf(fd,"%c",&n);


  do{
    fscanf(fd,"%c",&n);
  }while(!(n=='\n'));

  //EXTRACTION NX:
  do{
    fscanf(fd,"%c",&n);
    val*=10;
    val+=atoi(&n);
    //info=ajoutChar(info,n);
  }while(!(n==' '));
  //pic->nx=atoi(info);
  pic->nx=val/10;
```

```
  val=0;
  //resetStr(info);



  //EXTRACTION NY:
  do{
    fscanf(fd,"%c",&n);
    val*=10;
    val+=atoi(&n);
    //info=ajoutChar(info,n);
  }while(!(n=='\n'));
  //pic->ny=atoi(info);
  pic->ny=val/10;
  val=0;
  //resetStr(info);

  //EXTRACTION VALMAX:
  do{
    fscanf(fd,"%c",&n);
    info=ajoutChar(info,n);
  }while(!(n=='\n'));
  pic->valmax=atoi(info);
  resetStr(info);

  allocImage(pic);

  //EXTRACTION DES PIXELS
  for(iy=0;iy<pic->ny;iy++){
    for(ix=0;ix<pic->nx;ix++){

      fscanf(fd,"%c",&a);
      pic->tabl[iy][ix].r=(int)a;

      fscanf(fd,"%c",&a);
      pic->tabl[iy][ix].g=(int)a;

      fscanf(fd,"%c",&a);
      pic->tabl[iy][ix].b=(int)a;
    }
  }
  fermer(fd);
  return pic;
}
```

### 0.6.2 Finding the Digit or Character

```
rect delimit(image * pic){

  //Delimitation d un seul un caractere
  rect rectangle;

  //Coordonnees que l on veut trouver:
  int x1=-1, x2=-1, y1=-1, y2=-1;
  //iterateurs:
  int i,j;
  //
  int a;
  int b;
  a=0;
  b=0;


  //On parcours les colonnes de l image de haut en bas.
  //Au premier pixel noir qu on trouve on connait x1.
  for(i=0;i<pic->nx;i++){ // X
    for(j=0;j<pic->ny;j++){ // Y
      if(isWhite(pic,i,j)){
          x1=i;
          b=1;
          break;
        }
    }
    if(b==1){
      break;
    }
  }

  b=0;
  //On parcours les colonnes a droite de x1 de haut en bas.
  //A la premiere colonne entierement blanche on connait x2.
  for(i=x1+1;i<pic->nx;i++){ // Y
    a=0;
    for(j=0;j<pic->ny;j++){ // X
      if(isWhite(pic,i,j)){
        a=1;
      }
      if(a==0 && j==(pic->ny)-1){
        x2=i-1;
```

```
        b=1;
        break;
      }
    }
    if(b==1){
      break;
    }
  }

  b=0;
  //On parcours les lignes (de gauche a droite)
  //     dans lintervalle [x1;x2] pour trouver y1.
  //Au premier pixel noir quon trouve on connait y1;
  for(j=0;j<pic->ny;j++){
    for(i=x1;i<=x2;i++){
      if(isWhite(pic,i,j)){
        y1=j;
        b=1;
        break;
      }
    }
    if(b==1){
      break;
    }
  }

  b=0;
  //On parcours les lignes en dessous de y1
  //     dans lintervalle [x1;x2] pour trouver y2.
  //A la premiere ligne entierement blanche on connait y2.
  for(j=y1+1;j<pic->ny;j++){
    a=0;
    for(i=x1;i<=x2;i++){
      if(isWhite(pic,i,j)){
        a=1;
      }
      if(a==0 && i==x2){
        y2=j-1;
        b=1;
        break;
      }
    }
    if(b==1){
      break;
    }
  }
```

```
  rectangle.x1=x1;
  rectangle.x2=x2;
  rectangle.y1=y1;
  rectangle.y2=y2;

  return rectangle;
}
```

### 0.6.3 Formatting the Digit or Character to Fit Our Database

```
image * resize(image * pic, image * new){
  //Creation des donnees d une nouvelle image 28*28 pixels
  //a partir des donnees d une autre.

  new->ny=28;
  new->nx=28;
  new->entete=6;
  new->valmax=255;
  allocImage(new);
  new=initBlack(new);

  //Iterateurs:
  int k, h, i, j;
  //Nombre de pixel en hauteur du nouveau caractere:
  //int newheight = 24;
  //Nombre de pixel de l image de base pour
  //un pixel du nouveau caractere:
  int tilesize;
  //Nombre de pixel en largeur du nouveau caractere:
  int newwidth;
  //Moyenne d intensite pour un bloc de pixels de base
  //(donc valeur du pixel du nouveau caractere)
  int mean;
  int sideborder;


  tilesize = pic->ny / 24;
  newwidth = (pic->nx * 24)/pic->ny;
  sideborder = (28-newwidth)/2;

  for(k=0;k<pic->ny-tilesize;k+=tilesize){
    for(h=0;h<pic->nx-tilesize;h+=tilesize){
      mean=0;
      for(j=0;j<tilesize;j++){
        for(i=0;i<tilesize;i++){
          mean+=pic->tabl[k+j][h+i].r;
        }
      }
      mean/=(tilesize*tilesize);
      new->tabl[k/tilesize+2][h/tilesize+sideborder].r=mean;
      new->tabl[k/tilesize+2][h/tilesize+sideborder].g=mean;
      new->tabl[k/tilesize+2][h/tilesize+sideborder].b=mean;
    }
  }
```

```
  return new;
}
```

## 0.7 Structures and Functions

### 0.7.1 Images

```
struct rect{
  int x1;
  int x2;
  int y1;
  int y2;
};

struct triplet{
  int r;
  int g;
  int b;
};


struct image{
  int entete;
  int nx;
  int ny;
  int valmax;
  triplet ** tabl;
};
```

### 0.7.2 Matrices

**Here we have some examples of the matrix math library that we wrote:**

```
struct matrix{

  int mat_h;
  int mat_w;
  float ** mat;
};

matrix * matrix_sum(matrix * m1, matrix * m2, matrix * m3){

  int i, j;

  if((m1->mat_h != m2->mat_h)||(m1->mat_w != m2->mat_w)){
    printf("Somme␣des␣matrices␣impossible␣:
␣␣␣␣␣␣␣␣Les␣deux␣matrices␣n␣ont␣pas␣la␣meme␣shape\n\n");
  }
  assert((m1->mat_h == m2->mat_h)&&(m1->mat_w == m2->mat_w));
  m3->mat_h=m1->mat_h;
  m3->mat_w=m1->mat_w;
  alloc_matrix(m3);
  for(i=0;i<m3->mat_h;i++){
    for(j=0;j<m3->mat_w;j++){
      m3->mat[i][j]=m1->mat[i][j]+m2->mat[i][j];
    }
  }
  return m3;
}

void matrix_sigmoid(matrix * m){

  int i, j;

  for(i=0;i<m->mat_h;i++){
    for(j=0;j<m->mat_w;j++){
      m->mat[i][j]=1/(1+exp(-(m->mat[i][j])));
    }
  }
}
```

### 0.7.3 Artifical Neural Network

```
struct network{
  int nb_layer;
  layer * lyr;
  information info;
  int class;
};

struct layer{
  int nb_nrn;
  matrix * weights;
  matrix * outputs;
  matrix * error;
  float negative_target;
};


struct main_network{
  int nb_ntw;
  information info;
  float step_size;
  network ntw[26];
  bdd char_training[26];
  bdd * test;
  int batch_size;
  matrix * current_batch_data;
  matrix * current_batch_class;
};
```

## 0.8 Conclusion

In conclusion, this was a very entertaining and interesting project. We ran into difficulties and things did not always go the way we wanted, but that is a normal part of a project this complex. Among the things we would have liked to do (and certainly will in the future) are:

- Truly succeed at creating hidden layers
- Use gl4d to process images instead of our own functions
- Better master reading from and writing to files.
- Saving pre-trained weights was too difficult because of writing and (especially) reading very long floats
- Predict multiple digits and characters from the same image
- Predict digits and characters appearing in photos of handwriting