Technical documentation

Reading the datasets

A dataset is read from 2 files: one contains the pixel-data, the other one contains the labels. They are read simultaneously and stored in RAM in a Dataset struct.

Format of the image files

- 4B magic number
- 4B # images
- 4B # cols
- 4B # rows
- images –> 1B per pixel

Format of the label files

- 4B magic number
- 4B # labels
- labels –> 1B per label

Model breakdown

We use a fully-connected feed-forward deep neural network. A general case can be found in the image below.

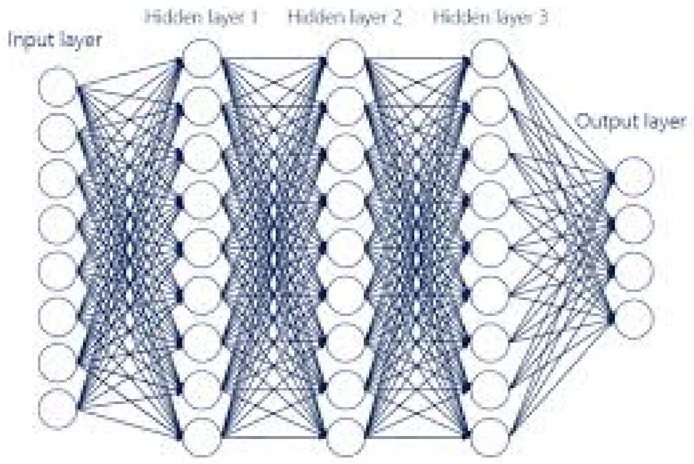

Figure 1: Image ofa neural network

Input layer

A flattened vector of the input pixels normalized to [0:1] interval.

Hidden layers

These are defined by the user. Their size and activation function can be customized.

Output layer

As this problem is a classification problem, the output layer's activation function is softmax. The output layer has 10 neurons, as we have 10 classes.

Implementing the feed-forward algorithm

For every layer, we compute ___f(W*x+b)___, where f is the activation function, W is the weight matrix, x is the previous layer's value-vector and b is the bias vector.

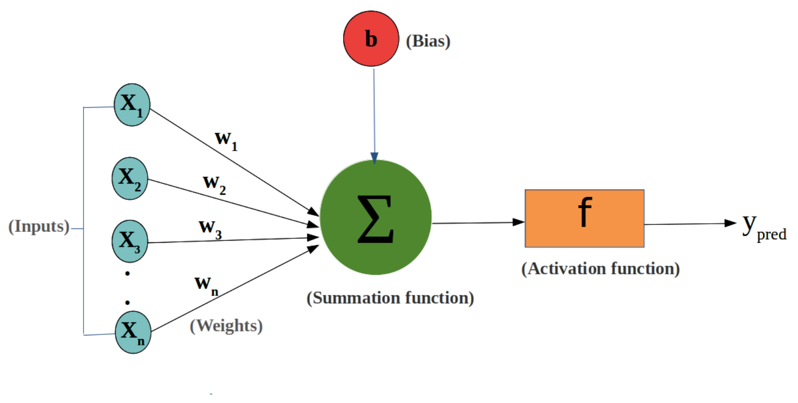

Figure 2: Image of the feed-forward algorithm

Implementing backpropagation

We use cross-entropy for the loss-function, as it is great for classification problems.

Articles for better understanding:

- Backpropagation
- Softmax and cross-entropy loss

We first compute the loss, then we compute the derivative of the loss with respect to the neurons in each layer. We do this first, so we can use the chain-rule later with ease.

Then we adjust the weights and biases using the chain rule. This is easy, as we just use the previously computed derivatives.

Overview of the program's structure

The program reads the arguments, determines the actions, it would have to take (training, testing of accuracy, showing images), and does them sequentially. The other parameters just customize this basic flow (eg.: by setting the dataset's folder). ## Saving / loading models

Each model can be saved/loaded to/from a file. Usually we use the .model extension, but it can differ.

The file-structure of the .model extension

- 1B # layers
- for each layer:
  - 4B inputDim
  - 4B outputDim
  - inputDim * outputDim * 8B weights
  - outputDIm * 8B biases

Important types

Model *(struct)*

Defines a model, holds its layers in a heap-allocated array.

Layer *(struct)*

Defines a layer, holds its dimensions, and its weights and biases in heap-allocated arrays.

Dataset *(struct)*

Defines a dataset, conatins its images and labels in a heap-allocated LabeledImage array.

LabeledImage *(struct)*

Defines an image with its label. Contains its pixels in a stack-allocated array. Should not be copied much!

Result *(struct)*

Defines a prediction-result. Holds 10 stack-allocated probabilities.

API documentation

dataset.h

ReadInt

```
int ReadInt (FILE* fp);
```

Summary: Reads 4 bytes from a file and converts them into a signed int32_t

ReadByte

```
uint8_t ReadByte (FILE* fp);
```

Summary: Reads 1 byte from a file

FreeDataset

```
void FreeDataset (Dataset dataset);
```

Summary: Frees up a dataset (with all its images)

GetPixelOfImage

```
double GetPixelOfImage(LabeledImage image, int x, int y);
```

Summary: Returns the pixel value of an image at the given coordinates

PrintLabeledImage

```
void PrintLabeledImage (LabeledImage image);
```

Summary: Prints the given image into stdout

ReadDatasetFromFile

```
Dataset ReadDatasetFromFile (const char* imagePath, const char* labelPath);
```

Summary: Reads 2 files (one for the images, one for the labels) and zips them into a single dataset

---

main.c

main

```
int main (int argc, char **argv) {
```

Summary: The entry point of the program.

---

manager.h

GetDefaultSetup

```
ProgramSetup GetDefaultSetup ();
```

Summary: Sets the program into default mode.

ProcessArgs

```
ProgramSetup ProcessArgs (int argc, char** argv);
```

Summary: Reads the command line arguments (passed as args here), and determines the mode, the program should be running in.

PrintImagesInfinitely

```
void PrintImagesInfinitely (Dataset dataset);
```

Summary: Prints images to stdout with their label for as long as the given dataset lasts

TestXORProblem

```
void TestXORProblem (
    int numNeuronsInHiddenLayer,
    ActivationFunction activationFunction,
    int iterations,
    double learningRate,
    double maxDeviationFromResult
);
```

Summary: Tests the integrity of the code, by running the XOR problem on it. If the backpropagation code is written successfully, the model should be able to learn a non-linear function (the XOR function).

GetAccuracy

```
double GetAccuracy (Model model, Dataset dataset);
```

Summary: Calculates the accuracy of a model on a dataset.

FitModel

```
void FitModel (
    Model model,
    Dataset trainSet,
    Dataset testSet,
    uint8_t numEpochs,
    double learningRate,
    bool saveContinuously,
    const char* savePath
);
```

Summary: Fits a model with the given parameters.

PrintImagesWithPredictions

```
void PrintImagesWithPredictions (Model model, Dataset dataset, bool onlyWrongs);
```

Summary: Prints images to stdout with their labels and the given model's predictions.

---

model.h

Sigmoid

```
double Sigmoid (double x);
```

Summary: The sigmoid function f(x) = 1 / (1 + e^-x)

SigmoidDer

```
double SigmoidDer (double x);
```

Summary: Derivative of the sigmoid function

ReLU

```
double ReLU (double x);
```

Summary: The ReLU function f(x) = x >= 0 ? x : 0

ReLUDer

```
double ReLUDer (double x);
```

Summary: The derivative of the ReLU function

LReLU

```
double LReLU (double x);
```

Summary: The Leaky ReLU function

LReLUDer

```
double LReLUDer (double x);
```

Summary: The derivative of the LReLU function

PrintModel

```
void PrintModel(Model model);
```

Summary: Prints a model to stdout

PrintModelLayout

```
void PrintModelLayout(Model model);
```

Summary: Prints a model's architecture / layout to stdout

PrintResult

```
void PrintResult(Result result);
```

Summary: Prints a result's probabilities to stdout

FreeModel

```
void FreeModel(Model model);
```

Summary: Frees up all the memory the given model uses (all weights, biases, layers, etc)

FreeLayer

```
void FreeLayer(Layer layer);
```

Summary: Frees up all memory the given layer uses

InitModelToRandom

```
void InitModelToRandom (Model* model, double randomRange);
```

Summary: Initializes the given model's weights and biases to a random value between [-randomRange, +randomRange] This will override all weights and all biases, so should only be called on uninitialized models!

CreateModel

```
Model CreateModel(int numHiddenLayers, ...);
```

Summary: Sets up the layers of a model from dimension and activation fn lists

- **numHiddenLayers**: number of hidden layers
- **...**: size, activation fn, size2, activation fn2, activation fn3 The number of additional parameters must be at least 1 (activation fn of last layer) @return model instance

CreateModelFromLayout

```
Model CreateModelFromLayout(LayerLayout* layout);
```

Summary: Creates an uninitialized model from a linked-list of LayerLayouts

MakeValueBufferForModel

```
double** MakeValueBufferForModel (int numLayers);
```

Summary: Allocates a buffer with dimensions defined by the model's layers. It does not create a buffer for the -1st vector (the image)

- **model**: @return pointer to the buffer

FreeValueBuffer

```
void FreeValueBuffer (Model model, double** buffer);
```

Summary: frees the value buffer

Predict

```
Result Predict(Model model, double* input, double** out_neuronValues);
```

Summary: Forwards the given image through the given model, and optionally saves the SUMS of the neurons to a buffer (not the values after the activation function)

- **model**: the model, with which we want to predict
- **input**: input image's data
- **out_neuronValues**: the SUMS!!! of the neurons, so sum(w*n + b)

Returns: the prediction result

CalculateCost

```
double CalculateCost(uint8_t label, double* resultValues);
```

Summary: implements cross-entropy algorithm cross-entropy: -sum(y[i] * log(s[i])) for i: 0..NUM_CLASSES

Returns: Cross-entropy cost of the given image with the label

BackPropagate

```
void BackPropagate(
    Model model,
    double** neuronValues,
    LabeledImage* image,
    double learningRate,
    double** preallocatedDerBuffer,
    double* out_cost
);
```

Summary: Adjusts the model's weights

- **neuronValues**: holds the values of the neurons, filled during the prediction phase

FitModelForImage

```
bool FitModelForImage(
    Model model,
    LabeledImage* image,
    double learningRate,
    double** preallocatedValueBuffer,
    double** preallocatedDerBuffer,
    double* out_cost
);
```

Summary: Runs predict and then backpropagation on the model with the given image and target.

- **model**: it is ok if it's not a pointer, because the layers variable will still point to the same memory address.
- **preallocatedDerBuffer**: a preallocated buffer to store the derivatives of the neurons. It is cheaper this way.
- **preallocatedValueBuffer**: a preallocated buffer to store the values of the neurons.

Returns: whether the result is ok or inf/-inf/nan –> true = good, false = stop learning.

CalculateAvgCostForModel

```
double CalculateAvgCostForModel (Model model, LabeledImage* images, int numImages);
```

Summary: Runs the model with the given image and gives back the cost

GetPredictionFromResult

```
int GetPredictionFromResult(Result result);
```

Summary: argmax

IsResultOk

```
bool IsResultOk (Result result);
```

Summary: Determines whether the result's probs are valid numbers.

SaveModelToFile

```
void SaveModelToFile (Model model, const char* filePath);
```

Summary: Writes the model's weights and biases to a file with the given path.

LoadModelFromFile

```
Model LoadModelFromFile (const char* filePath);
```

Summary: Reads the given file and returns the model stored in it.