Assignment 2: Machine Learning Project

Topic 10: Neural networks

Innholdsfortegnelse

Content 2

Assignment answer: 3

Part 1 3

Sunglass recogniser: 3

Face recogniser: 3

Pose recogniser: 4

Part 2: 5

Part 2.1 5

Part 2.2 5

Self written documentation 9

Faces: 9

Code: 9

Facetrain 10

Sources: 11

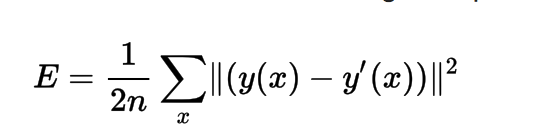
# 

# Part 1 – getting to know the code

## Sunglass recogniser:

* What did I change:
  + The if statement in imagenet.c
  + Instead of comparing userid to “Glickman”, it is now comparing eyes to “sunglasses”.
* Training set:
  + Maximum accuracy achieved was 100%.
  + This happened at epoch 30.
* Validation set:
  + Maximum accuracy achieved was 100%.
  + This happened at epoch 20.
* Training set:
  + Maximum accuracy achieved was 94.2308%.
  + This happened at epoch 45.

## Face recogniser:

* What did I change:
  + I change imagenet.c to add an appropriate target vector to each image.
    - Implemented a for loop which iterates through all of the names
  + Then I changed the evaluate performance () method to iterate through all of the targets and see if it guessed correctly.
  + Created a method output\_result\_on\_imagelist ()
    - This outputs for each image in the list if the function guessed correctly.
  + Error function changed to
  + 
* Training set:
  + Maximum accuracy achieved was 100%.
  + This happened at epoch 54.
* Validation set:
  + Maximum accuracy achieved was 94.4%.
  + This happened at epoch 92.
* Training set:
  + Maximum accuracy achieved was 87.5%.
  + This happened at epoch 69.
* As you can see from the image wrongly\_guessed\_images, all of them are very similar, and that there is little distinguishing them from each other.
* Face recognizer run on full set with full resolution **(**tested later**):** 
  + Train set:
    - Maximum classification on epoch 93, reaching 98.917%
  + Test set 1:
    - Classification accuracy: 92.8058% #### Number of wrong guesses: 10 #### average of the error function: 0.06729%
  + Test set 2:
    - Classification accuracy: 92.3077% #### Number of wrong guesses: 16 #### average of the error function: 0.0735408%

## Pose recogniser:

* I encoded the output the same way, only with focus on poses instead of userid.
* Training set:
  + Maximum accuracy achieved was 99.639%.
  + This happened at epoch 48.
* Validation set:
  + Maximum accuracy achieved was 85.6115%.
  + This happened at epoch 99.
* Training set:
  + Maximum accuracy achieved was 91.3462%.
  + This happened at epoch 51.
* Weight tuning discussed:
  + I did two training sessions before looking at the hidden units weight. One training session on full resolution and one on a quarter of the resolution.
  + You can find the images showing the weights in the folder “weights of hidden units”.
  + The hidden units are definitely weighing particular parts of the image differently. If you compare unit 1 and unit 3 for example, then you can see that they are almost the same, only reversed. (left side weights of unit 1 has the same transformation as right side of unit 3).
  + Comparing different units, we can see that some is focusing on the difference of the pixels compared (edges). While some are focused on the concentration.

# 

# Part 2 – Improving performance of the network

As my part two of this assignment, I wanted to convert the neural network into a convolutional neural network. I worked alone on the part two, due to the task I wanted to do. Unfortunately, I didn’t manage to complete the task in time, due to lack of time and programming skills in C. What I did instead was to pivot my project and do a lot of test on the network with varying hidden units, training examples, learning rate and momentum. I did this to see what effect it would have.

I will split the part two into two parts; one explaining what I wanted to do, and what effect that would have had, secondly, I will discuss what effect the changes, I did on the values mentioned, had.

## Part 2.1 – Convolutional neural network for deep learning

### Intro

In my original task, I wanted to implement a set of kernels (filters) that I would send the image through. This would for each layer create x set of input values that in some way was focusing on a specific attribute or feature of the image (i.e. edges). Then I wanted to send these images into the fully connected feed forward layer to create 5 different output-units arrays. Then I wanted to send these into a smaller fully connected layer output the final decision. To minimize the work for the neural network and to avoid over fitting I would also implement pooling layers.

**Convolution**

**Pooling**

**Deep learning**

## Part 2.2 – testing the network

### Intro

After pivoting the assignment I wanted to see how different parameters would affect the network. I therefor used the facial recognizer and slightly changed it to implemented a different output. Then I tested the network on the training set with varying parameters set. Parameters that were changed where: learning rate, momentum, numbers of neurons in the hidden layer, and training list size.

In the coming sections shall I take a look at some of the possible problems with the back propagation algorithm, discuss what causes it and what you can do to prevent it.

### Problem

**Local Minima**

* In gradient descent we start at some point on the error function defined over the weights, and attempt to move to the global minimum of the function. In a perfect model, like the one visualised, any step in a downward direction will take us closer to the global minimum.
* For real problems, however, error surfaces are often more complex, and is more likely to look like the figure to the right. Here there are many local minima, and the ball is shown trapped in one such minimum. Progress here is only possible by climbing higher before descending to the global minimum.

**Overfitting**

* This is when the model describes random error or noise instead of the underlying relationship of the data. This means it will have good performance on the training data and poor generalization to other data.
* Overfitting often happens when a model is too complex in comparison to the amount of the data.
* Your model is overfitted when you have a high variance

**Underfitting**

* When the model fails to catch the underlying trend of a dataset, which results in poor performance on the training data and poor generalization to other data.
* Your model is underfitted when you have a high bias.

### Learning rate

Learning rate is applied to make it possible to determine how fast or slow we will move towards the optimal weight. In order for back propagation to work in a good manner we must set the learning rate to an appropriate value. If the learning rate is very large we will skip the optimal solution. If it is too small we will need too many iterations to converge to the best values. So using a good learning rate is crucial.

**Learning rate adaption**

To minimize learning time and to maximise the effectiveness of the learning, you can update the learning rate each iteration, instead of having it constant. The idea behind this is that the farther you are from optimal solution, the faster you should move towards it, and thus the learning rate should be larger. The closer you get to the solution the smaller it should be. Unfortunately since you don’t know the actual optimal solution, you also don’t know how close you are to them in each step.

A solution is to anneal (gradually lower) the global learning rate. A simple, non-adaptive annealing schedule for this purpose is the search-then-converge schedule given by:

µ (t) = µ(0)/(1 + t/T); where µ is the learning rate and t is time or epoch.

This technique is allowing the network to find the general location of the minimum, before annealing it at a very slow pace that is known from theory to guarantee convergence to the minimum. The time T of this schedule is a new free parameter that must be determined by trial and error.

### Momentum

A technique that can help the network out of local minima is the use of a momentum term. The way this affects the model, is that it always add a proportion, m (momentum), of the previous weight update, to the new one.

### The Number of Neurons in the Hidden Layers

Deciding the number of neurons in the hidden layer is a very important part of deciding your overall neural network architecture, and it can easily make a big difference to the performance and the result.

Using too few neurons in the hidden layer will result underfitting. This happens because there are too few neurons in the hidden layer to adequately detect the signals in a complicated data set.

Using too many neurons in the hidden layer can result in two big problems.

First, too many neurons in the hidden layer may result in overfitting. This happens because the neural network has so much information processing capacity that the limited amount of information contained in the training set is not enough to train all of the neurons in the hidden layers.

Second, A large number of neurons in the hidden layer can increase the time it takes to train the network. The amount of training time can increase to the point that it is impossible to sufficiently train the neural network. Obviously, some compromise must be reached between too many and too few neurons in the hidden layer.

Some common rules-of-thumb for determining the number of hidden units are:

* The number of hidden neurons should be between the size of the input layer and the size of the output layer.
* The number of hidden neurons should be 2/3 the size of the input layer, plus the size of the output layer.
* The number of hidden neurons should be less than twice the size of the input layer.

### The testing

I did several tests on the full network to see how it would respond. Some of the test had to be run on a smaller network due to time complexity. I have recorded some of the test that I found interesting and that I wanted to discuss.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 80 images in training set  36 images in test1 set  40 images in test2 set | | | | |
| **Test** | **Momentum** | **Learning rate** | **Nr of hidden units** | **Result to notice** |
| 1 | 0.8 | 0.8 | 20 | Never converged |
| 2 | 0.5 | 0.3 | 20 | Epoch: 33  Result on train list: 92.5%  Result on test list: 72.2% |
| 3 | 0.8 | 0.8 | 2 | Epoch: 846  Result on train list: 5%  Result on test list: 2.8% |
| 4 | 0.3 | 0.3 | 1820 | Epoch: 2784  Result on train list: 5%  Result on test list: 2.8% |
| 5 | 0.3 | 0.3 | 4 | Epoch: 4798  Result on train list: 100%  Result on test list: 66% |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 277 images in training set  139 images in test1 set  208 images in test2 set | | | | | |  |
| Test | Mo-mentum | Learning rate | Epoch it Reached >95% accuracy on test list | Validation accuracy | Training accuracy | Note |
| 6 | 0.3 | 0.3 | 177 | 93.75 | 98.92 |  |
| 7 | 0.3 | 0.8 \* 1/(1 + 0.05 \* epoch) | Reached 61% on epoch 125, manually stopped at epoch 227 since it was still on 61% (local minimum) | --- | 66% |  |
| 8 | 0.3 | 0.3 \* 1/(1 + 0.01 \* epoch) | Reached 86% at epoch 43. Manually stopped at 141 after staying at 84% for almost 100 epochs. |  | 88% |  |
| 9 | 0.5 | 0.3 \* 1/(1 + 0.01 \* epoch); | Reached 80% at epoch 38. Manually stopped at 161 after diverging to 79%. | --- | 85% |  |
| 10 | 0.5 | 0.3 \* 1/(1 + 0.001 \* epoch); | Reached 87% at epoch 39. Manually stopped at epoch 205 at 84. Had many local minima before this. |  | 81% | Model where the test list had better accuracy almost through the whole test |

Too see the full test result from these files, check out the facial-recognition folder.

### Discussion

Considering the three problems mentioned at the beginning, we can now classify the test result:

**Local minima**

From test 7, 8, 9 and 10 it looks like the model hit a local minima. For all of them, there was almost, if not more than 100 epochs without any progression. In test 7, the learning rate converged to fast, and thus, I think it got stuck. From test 8 and 9 you can see a big improvement from test 7. I changed the rate, which the learning rate was converging on, so that it would go 5 times slower. On test 9 and 10 I also raised the momentum, and the result started to go more up and down around 80%. One reason for this might be that the momentum was too big. You can especially see on test 10 that it has more variation than test 9. The difference between these two was that the learning rate converged faster on test 9.

**Overfitting and Underfitting**

As mentioned earlier, the number of neurons in the hidden layer has a lot to say. But this is where my test result didn't support the theories found.

First of all, test 5 should be a good example of underfitting the network. The net only contains 4 hidden neurons, which is way to little compared to the 960 input units and the 20 output units.

However, for backprop with a constant learning rate, the learning rate must be set small enough to avoid divergence in the ill-conditioned regions of the error surface.

# Self written documentation

## Faces:

* 20 directories
  + One per person – named by user ID
* Naming convention - userid\_pose\_expression\_eyes\_scale.pgm
  + User ID:
    - 20 different values (number of persons)
  + Pose:
    - Head position of the person
    - 4 values – straight, right, left, up
  + Expression
    - 4 values – neutral, happy, sad, angry
  + Eyes
    - 2 values - open or sunglasses
  + Scale
    - How big is the image
      * 1: 128cx120r
      * 2: 64x60
      * 4: 32x30
    - Will be using the quarter resolution (4) for the training to spare time
  + NB: some images have a bad suffix. Meaning they contains glitches.

## Code:

* **Files not to modify:** 
  + Pgmimage.c – read and write of pgm image files
    - Data structures:
      * IMAGE and IMAGELIST (array of pointers to IMAGE)
  + Backprop.c – neural network package.
    - Supports three layer fully connected feed forward network.
    - Uses back propagation algorithm for weight tuning
    - Routines for creating, training and using networks.
  + Hidtopgm.c
    - Hidden unit weight visualization utility
    - Interesting to explore some of the possible alternate visualization schemes.
* **Files to modify:** 
  + Imagenet.c
    - Interface routines for loading images into the input units of a network and setting up target vectors for training.
    - Modify load\_target when implementing the face and pose recognizer, so that it contains appropriate target vectors!
  + Facetrain.c
    - The top-level program that uses all of the other modules.
    - Modify to change network sizes and learning parameters
    - Also modify performance\_on\_imagelist () and evaluate\_performance () for the face and pose recognizer.

## Facetrain

* **Running**:
  + ./facetrain –n <network file> -e <number of epochs> -T <test only> -s <seed> -S <number of epochs between saves> -t <training image list> -1<test set 1 list> -2 <test set 2 list>
  + Network file – loads or creates a file to save the network
  + Number of epochs – specifies number of training epochs (default = 100)
  + Test only – performance reported on all three sets.
  + Seed – input used as seed for random number generator (default = 102194)
    - Allows you to reproduce experiments if necessary
  + Number of epochs between saves – default= 100, thus, if you train for 100 epochs, then the document is only saved after training is completed.
  + Training image list
    - If this option is not specified, it is assumed that no training will take place epochs = 0.
    - In this case the statistics for the training set will all be zeros
  + 2 test sets serves for the purpose of having one set to train and test on, and when the performance on the test set begins to degrade you can use the second set for a “real” test.
* **Output:**
* For each epoch the following performance measures are output
  + **Epoch delta trainperf trainerr t perf t err t perf t err**
* **Epoch:** number of epoch just completed
* **Delta:**  sum of all delta values on the hidden and output units as computed during back propagation, over all training examples for that epoch.
* **Trainperf:** Percentage of examples correctly classified in the training set
* **Trainerr**: Average, over all training examples, of the error function
  + 0.5\*sigma (ti – oi)^2
* **T1perf**: percentage of examples in test set 1 correctly classified
* **T2err:** Average, over all training examples, of the error function
* **T2perf: --“”---** set 2.
* **T2err**: ---“”--- set 2.

# Sources:

* Sparse autoencoder

<http://web.stanford.edu/class/cs294a/sparseAutoencoder_2011new.pdf>

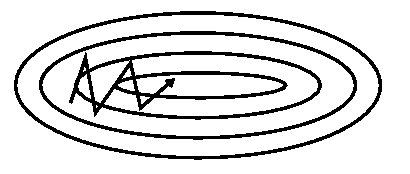
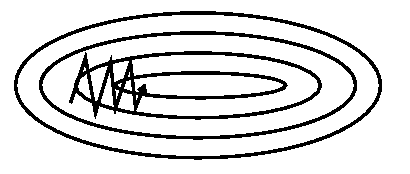
* No More Pesky Learning Rates

<https://arxiv.org/pdf/1206.1106.pdf>

* Momentum and Learning Rate Adaptation

<http://www.willamette.edu/~gorr/classes/cs449/momrate.html>

When the gradient keeps changing direction, momentum will smooth out the variations. This is particularly useful when the network is not well-conditioned. In such cases the error surface has substantially different curvature along different directions, leading to the formation of long narrow valleys. For most points on the surface, the gradient does not point towards the minimum, and successive steps of gradient descent can oscillate from one side to the other, progressing only very slowly to the minimum (Fig. 2a). Fig. 2b shows how the addition of momentum helps to speed up convergence to the minimum by damping these oscillations.



<http://www.willamette.edu/~gorr/classes/cs449/momrate.html>

In most Supervised Machine Learning problems we need to define a model and estimate its parameters based on a training dataset. A popular and easy-to-use technique to calculate those parameters is to minimize model’s error with Gradient Descent. The Gradient Descent estimates the weights of the model in many iterations by minimizing a cost function at every step.