**Object Detection/Recognition for Haofu**

**Goal :** *With a technician wearing a portable head camera, be able to recognize which tool he is carrying.*

**The process is separated into two parts :**

Contents

[1. Object detection 1](#_Toc487529846)

[Color: 1](#_Toc487529847)

[Image tracking : 2](#_Toc487529848)

[2. Image recognition 3](#_Toc487529849)

## Object detection

In order to detect the object carried, the hand must be detected. This is a background subtraction application, with a moving background. We wanted to combine two methods : **Color Selection** and Image Tracking

### Color:

1. Color selection
2. Contour detection
3. Selection of the contours touching the bottom part of the image (other contours cannot be the user's hand)
4. Selection of the biggest remaining contour
5. Approximation of the contour
6. Estimation of the position+size of the object according to the position+size of the hand.

|  |  |  |
| --- | --- | --- |
| Figure 1 : Original video (the colors change are of no importance) | Figure 2 : Color selection | Figure 3 : Contour detection/selection + poly approximation + center |

*Results :*

As this is only first testing, no quantification of the results has already been done. The observations following are purely qualitative.

The results are not so bad, however, of course, when objects of the same color appear, such as wooden furniture, the results are not very good, even with the contour selection.

**Hand/ Gesture Recognition:**

1. Segment the hand skin by detecting skin color
2. Get n biggest contours
3. Describe contours with convex polygons and find convexity defects
4. Dismiss contours not representing hands
5. Separate a certain area around the hand for further tool recognition

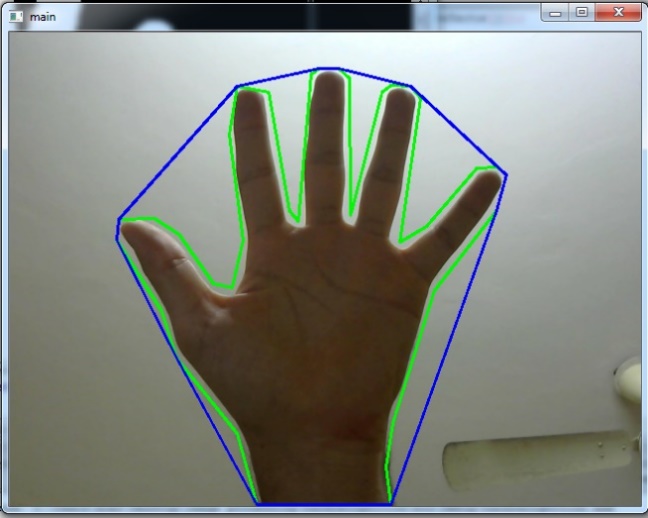


Figure 5: example of hand detection

*Results:*

It is the next step we are going to try, which is able to optimize the skin color detection when other objects have the similar color, so that we can get a more ideal focusing area for tool recognition.

### Image tracking :

Tracking important features in the video. The points belonging to the background should have a linked movement (rotation + translation). The other points belonging to the hand+object will have a different movement.

Separating the hand+object from the background.

*Using openCV Lucas-Kanade optical flow method.*

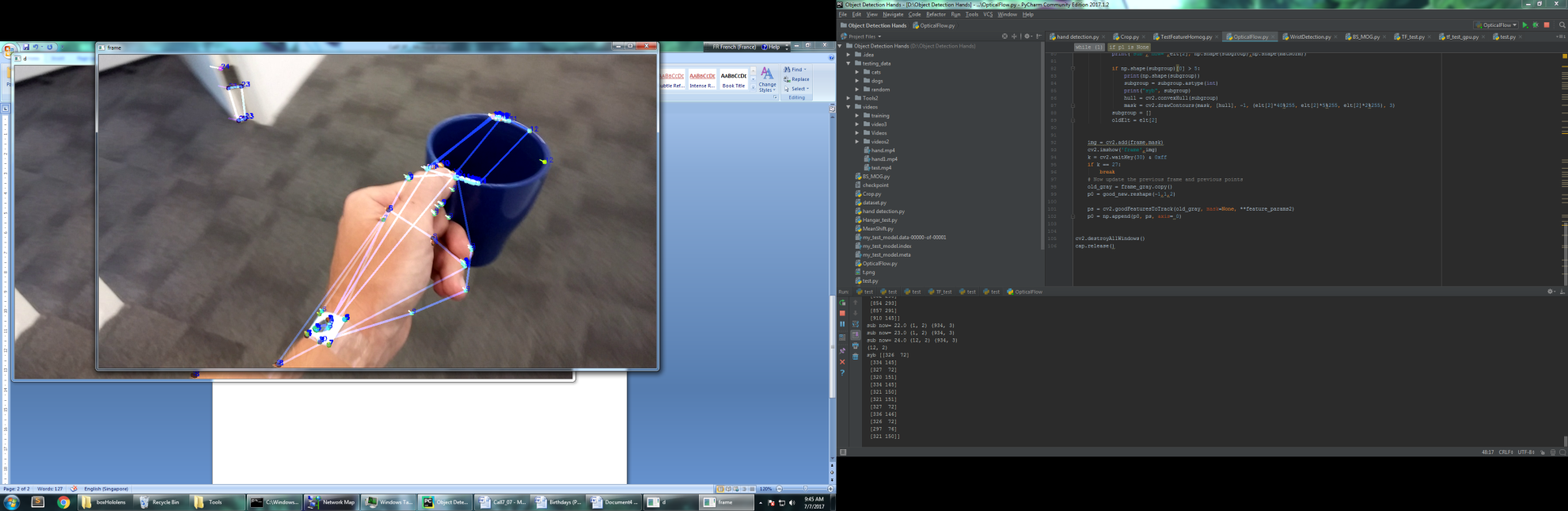


Figure 4 : Quick test connecting the points with same movement vector norm

*Results :*

Currently, we have no quantified results with this method, only because it is not finished. To continue, we need to :

1. Estimate the global movement of the camera (translation + rotation)
2. Separate the hand+object from the background

## Image recognition (Deep Learning)

**Part 1:**

First try using the open source library TensorFlow from Google. Adaptation of an online tutorial making the difference between cats and dogs, to try to differiencate 6 tools, using CNN.

The tutorial architecture is :

* 3 convolutional layers (the filter is a random normal distribution)(We tried adding and removing layers, the results were better with more layers)
* 1 flattening layer
* 1 fully connected layer

|  |  |  |  |
| --- | --- | --- | --- |
| D:\Object Detection Hands\Tools2\testing_data\blueScissors\blueScissors.1.jpg | D:\Object Detection Hands\Tools2\testing_data\boxHololens\boxHololens.20.jpg | D:\Object Detection Hands\Tools2\testing_data\greenScrewdriver\greenScrewdriver.21.jpg | D:\Object Detection Hands\Tools2\testing_data\pinceVerte\pinceVerte.39.jpg |

Figure 5 : Examples of the tools

**Process :**

1. We took some still videos of us holding different tools, on the same white background.
2. From the videos, we extracted the frames (tried with 200, 500 and 2000 images)
3. We fed all the data to the model, and trained it, with :
   1. Original Frames
   2. Frames after background subtraction
   3. Contour detection frames (we had the best results)
4. We tried the result on an other video showing every tools one after the other.

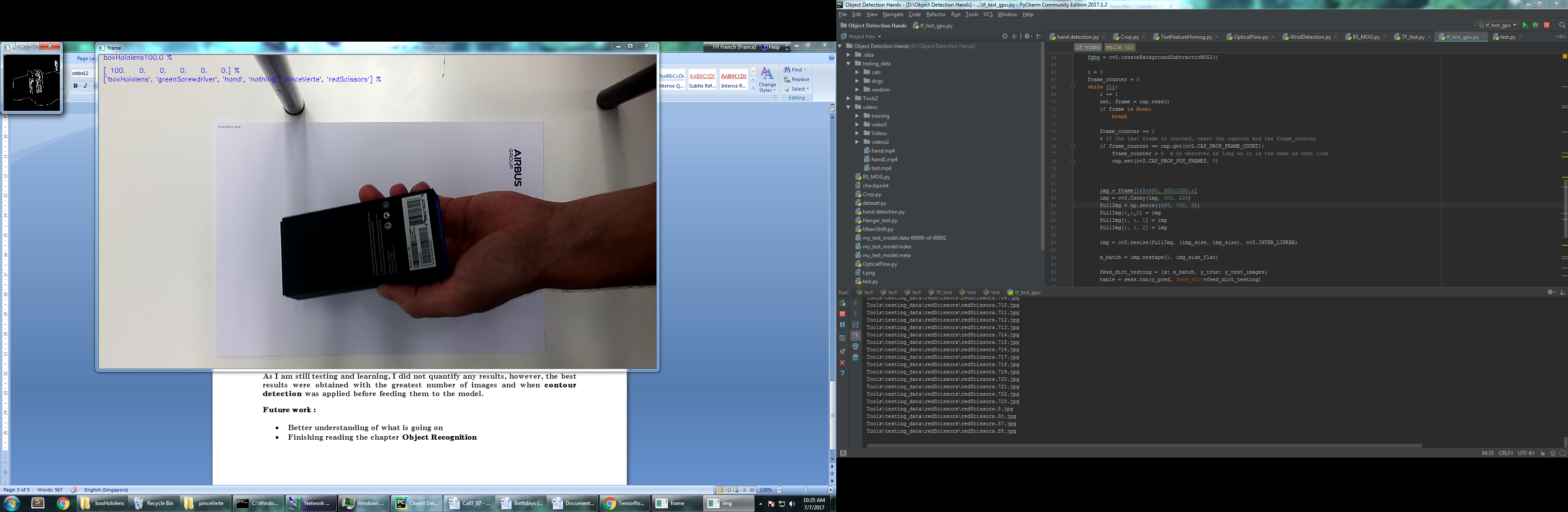


Figure 6 : Video test, the name of the object detected is in the top left hand corner

As we are still testing and learning, we did not quantify any results, however, the best results were obtained with the greatest number of images and when **contour detection** was applied before feeding them to the model.

Roughly speaking, 5 tools were detected out of 6, for the best model.

**Future work:**

* Better understanding of what is going on.
* Try to avoid overfitting, which might be the current issue.
* Build a dataset, using real and synthetic images.

**Part 2:**

**Concepts**

**Image classifier:**

* When no obvious way to hard-code the algorithm for recognizing a cat, or other classes.

So:

* Data-driven approach:

1. Collect a dataset of images and labels

2. Use Machine Learning to train an image classifier

3. Evaluate the classifier on a withheld set of test images

**NN classifier (k-nearest neighbour) :**

* Remember all training images and their labels, and predict the label of the most similar training image
* Use distance matrix to compare test images and training images, find pixel-wise absolute value differences. The lowest is classified into the same class.
* Choice of distance: hyperparameter (eg. Manhattan distance, Euclidean distance).

How to choose: tuned using a validation set, or through cross-validation if the size of the data is small

* k is also a hyperparameter to show no. of classes
* problem: never used to classify images. Cannot detect for any shift, messed up or light adjustment.

10\*3072 3072\*1

---Example of MMUL

10\*1

**Linear Classifier:**

W: Weight

b: Bias

x: Determined by photo

**Loss Function:**

* **Hinge loss-- (Multiclass) SVM loss**

Given an example where is the image and where is the (integer) label, and using the shorthand for the scores vector

: score vector of the correct class

The larger the worse

**Score Function:**

**Full Training Loss**

: regularization strength (hyperparameter)

* **Cross-entropy loss-- Softmax Classifier** (Multinomial Logistic Regression)

scores = unnormalized log probabilities of the classes.

Where

0.13

0.87

0.00

24.5

164.0

0.18

3.2

5.1

-1.7

exp🡪 normalize🡪 🡪

Provided the 1st class is the correct class

The closer to 0, the better.

**Optimization:**

Follow the slope. In multiple dimensions, the gradient is the vector of partial derivatives

The loss is a function of W

* Numerical gradient: approximate, slow, easy to write
* Analytic gradient: exact, fast, error-prone
* In practice: Always use analytic gradient, but check implementation with numerical gradient. This is called a **gradient check**.

**Back Propagation** All partial derivative

Loop:

1. Sample a batch of data

2. Forward prop it through the graph, get loss

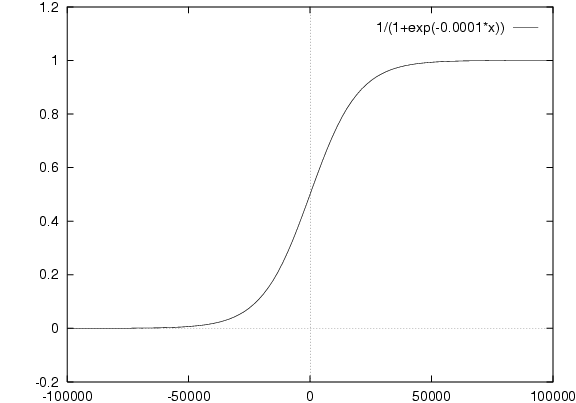
3. Backprop to calculate the gradients

4. Update the parameters using the gradient

**Activation Function:**

Ideal features: 1. Range [-1,1]

2. Zero centered

3. Doesn’t saturate

* **Sigmoid**

Problem: 1. Saturated neurons “kill” the gradients

2. Sigmoid outputs are not zero-centered

3. exp() is a bit compute expensive



* **Tanh(x)**

Good: 1. Squashes numbers to range [-1,1]

2. zero centered (nice)

Problem: 1. still kills gradients when saturated

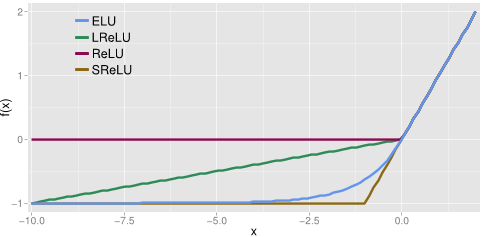
* **ReLU**(Rectified Linear Unit)

Good: 1. Does not saturate (in +region)

2. Very computationally efficient

3. Converges much faster than sigmoid/tanh in practice

Problem: Not zero-centered output

* People like to initialize ReLU neurons with slightly positive biases(e.g. 0.01)
* X<0: dead ReLU. Will never activate/ never update

Good: will not die if x<0

* **PReLU**
* **ELU**

Good: Closer to zero mean outputs

Not so good: Computation requires exp()

* **Maxout Neuron**

Good: Generalizes ReLU and Leaky ReLU

Not so good: Doubles the number of parameters/neuron

In practice:

Use ReLU. Be careful with your learning rates

Try out Leaky ReLU / Maxout / ELU

Try out tanh but don’t expect much

Don’t use sigmoid

**Batch Normalization:**

* Improves gradient flow through the network
* Allows higher learning rates
* Reduces the strong dependence on initialization
* Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

In practice: use

**Weight initialization**

W=np.random.randn(fan\_in,fan\_out\*1.0

W=np.random.randn(fan\_in,fan\_out)/np.sqrt(fan\_in)

W=np.random.randn(fan\_in,fan\_out)/np.sqrt(fan\_in/2)

In practice: use Xavier initialization

**Parameter Updates**

* **A Simple Update (SGD):**

x += -learning\_rate \* dx

* **Momentum Update:**

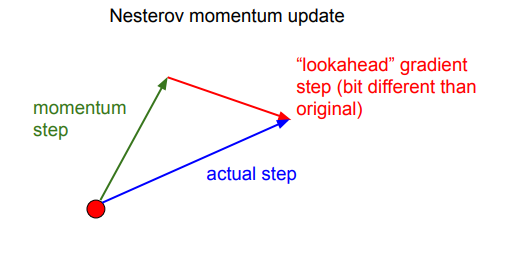
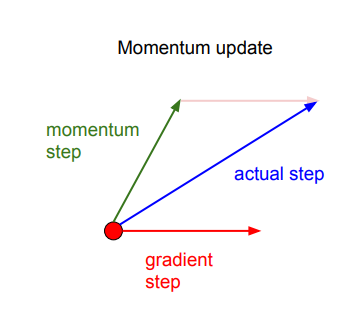
v = mu \* v – learning\_rate \* dx (momentum step+ gradient step)

x += v

mu: usually 0.5, 0.9 or 0.99

May overshoot but will go back

* **Nesterov Momentum update (nag: Nesterov Accelerated Gradient)**

****

v\_prev = v

v = mu \* v – learning\_rate \* dx

x += -mu \* v\_prev + ( 1 + mu ) \* v

* **AdaGrad Update**

cache += dx\*\*2

x += -learning\_rate \* dx / (np.sqrt(cache) + 1e-7 )

* **RMSProp update**

cache = decay\_rate \* cache + (1-decay\_rate) \* dx\*\*2

x += -learning\_rate \* dx / (np.sqrt(cache) + 1e-7 )

* **Adam Update**

Look like RMSProp with momentum + bias correction

In Practice: Adam is a good default choice in most cases

**Dropout**

p = 0.5 # probability of keeping a unit active. Higher=less dropout

def train\_step(x):

U1= (np.random.rand(\*H1.shape) < p) / p # first dropout mask

H1 \*= U1 # drop

U2= (np.random.rand(\*H2.shape) < p) / p # first dropout mask

H2 \*= U2 # drop

out=……

def predict(x):

H1 = np.maximum(0, np.dot(W1, X) + b1)

H2 = np.maximum(0, np.dot(W2, H1) + b2)

out=……

**CNN**

* e.g. 32x32x3 image + 5x5x3 filter 🡪 28x28x1 activation map

**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”

32x32x3 image + 6 5x5x3 filter 🡪 28x28x6 activation map

28x28x6 activation map + 10 5x5x6 filter 🡪 24x24x10 activation map

* Image size=N, filter size=F, stride=m: output size= (N-F)/stride +1 (must be exact division)
* If **pad** (stride 1 pad with 1 pixel border), usually zero pad with (F-1)/2, to prevent the size become smaller after every layer
* Number of **filter**: usually exponent of 2
* **Number of parameters**: all number time together
* **Typical architectures** look like [(CONV-RELU)\*N-POOL?]\*M-(FC-RELU)\*K,SOFTMAX, where N is usually up to ~5, M is large, 0 <= K <= 2.
* **Pooling layer: -** down sampling

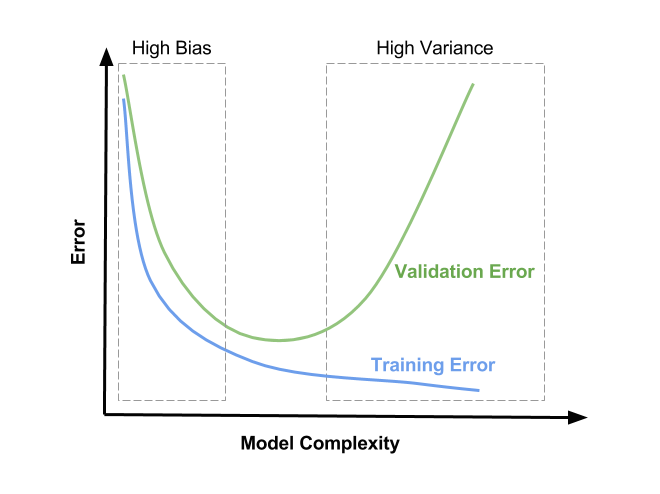
**-**makes the representations smaller and more manageable

- operates over each activation map independently

e.g. 224x224x64 🡪 after pooling: 112x112x64

* **Fully connected layer** (FC layer): -Contains neurons that connect to the entire input volume, as in ordinary Neural Networks
* One **epoch** = one forward and one backward pass of all training examples
* **Batch size** = number of training examples in one forward/ backward pass
* **Number of iterations** = number of passes, each pass using [batch size] number of examples

**Ensemble Learning**

**Boosting**: -Train learners sequentially, focus new learners on examples that others get wrong

- Learn a regression predictor 🡪 compute error residual 🡪 learn to predict the residual

-Bootstrap: create a random subset of data by sampling.

**Bagging**: -learn many classifier, each with only part of the data

-combine them through averaging 🡪 reduce overfitting

**Biases**: Error due to model choice

**Variance**: Randomness due to data size 🡪 very dependent on the data we use to train

AdaBoost, GBDT, Random Forest, Hybrid, Diversity, sklearn

**Model**

**Dataset**

We took videos of different people holding different tools. Tools are red and blue scissors, large and small wire cutters, red and green screwdrivers. Every video around 10-30 seconds. We extracted the video frame by frame and resize them using the script VideoTools.py. After resizing to different sizes and saving into different folders, we randomly classify the images to training\_data and testing\_data. There are around 30,000 photos in training data and 700 in testing data. In each dataset folder, we classify the images into 6, 4 or 3 folders, according to the need. For instance, we don’t necessarily need to recognize the colour when the model is distinguishing if the technician is using the correct tool, so that we combine two different colour of tools into one, which gives us 3 or 4 classes. But when reminding the technician of any leftover tools after finishing a task, it is better to know which exactly is the one. In this case we train the model to 6 classes.

Script to extract video: D:\Object Detection Hands\Background Subtraction and CS\VideoTools.py

Images paths: 512x512, 3 classes D:\Object Detection Hands\Tools

128x128, 3 classes D:\Object Detection Hands\Toolss

128x128, 4 classes D:\Object Detection Hands\Tools4classes

128x128, 6 classes D:\Object Detection Hands\ToolsSD

719x719, 3 classes D:\Object Detection Hands\Tools6

Video path: D:\Object Detection Hands\videos\VideosDataset

**Main Code**

* dataset.py and dataset2.py

-Load and read training and testing images from folder with Opencv.

-Get the number of images, their labels, classes, index, and define epoch.

-Shuffle the images once they are loaded.

* TF\_test.py

-Get the training and testing set from dataset.py.

-Use tensorflow

-Several helper functions to initialize weight and biases, define CONV and FC layers, flatten layer. Optimization.

-Write to excel to record details in each training, accuracy after every epoch, time spent.

-Save the session to file.

* tf\_test\_gpu.py

-Load the session saved before.

-Load the testing data (testing images or video) and feed into the session.

-Print testing accuracy. Show wrong images

-Save the result to excel.

**Results**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Image size | 128 | | 128 | 128 | 128 | 128 | 128 | | 512 | 512 | 512 | 128 |
| No. of classes | 6 | | 6 | 6 | 4 | 3 | 3 | | 3 | 3 | 3 | 3 |
| No. of channels | 3 | | 3 | 3 | 3 | 1 | 3 | | 3 | 3 | 3 | 3 |
| No. of conv layers | 2 | | 4 | 4 | 4 | 4 | 4 | | 4 | 4 | 4 | 4 |
| Learning Rate | 1e-4 | | 1e-4 | 1e-4 | 1e-4 | 1e-3 | 1e-3 | | 1e-3 | 1e-3 | 1e-3 | 1e-3 |
| Batch Size | 64 | | 64 | 64 | 64 | 64 | 1024 | | 64 | 256 | 512 | 2048 |
| No. of iterations | 1600 | | 1600 | 6400 | 30476 | 6400 | 10000 | | 6400 | 1000 | 1000 | 1000 |
| Train Accuracy | 93.75% | | 89.06% | 100% | 100% | 100% | 100% | | 100% | 100% | -- | -- |
| Test Accuracy | **70.62%** | | **66.36%** | **72.90%** | **84.42%** | **~50%** | **89.07%** | | **82.15%** | **87.23%** | **--** | **--** |
| Validation Loss | 0.0986 | | 0.5234 | 0.0329 | <1e-4 | <1e-4 | <1e-4 | | -- | <1e-4 | -- | -- |
| Comment | Layer not enough | Need more data to train if more layers | | More iterations | | Greyscale less data, bad performance | | Best result so far | Higher resolution is better |  | OOM | OOM |

**Improve Performance**

* With data:

- get more data

- invent more data (data augmentation/ data generation)

- transform data

- feature selection

* With algorithm:

- spot-check: tree methods (random forest, gradient boosting), instance methods (SVM, kNN)

- Resampling

* Algorithm tuning:

- Weigh initialization

- learning rate

- activation function

- batches and epochs size

- optimization and loss (e.g. Adam)

- early stopping

- network topology (wide/ deep layer)

* Ensembles

e.g. training accuracy>>validation accuracy 🡪 maybe overfitting 🡪 regulation

training and validation accuracy both low 🡪 underfitting 🡪 train more

inflection point when training goes above validation accuracy 🡪early stopping